BONVOYAGE

From Bilbao to Oslo, intermodal mobility solutions, interfaces and applications for people and goods, supported by an innovative communication network

Research and Innovation Action GA 635867

Deliverable D5.2:

Development and test of the adaptation functionality

Deliverable Type:	Report
Deliverable Number:	5.2
Contractual Date of Delivery to the EU:	15.09.2017
Actual Date of Delivery to the EU:	15.09.2017
Title of Deliverable:	Development and test of the adaptation functionality
Work package contributing to the Deliverable:	WP5
Dissemination Level:	PU
Editor:	SINTEF

Author(s):	ATOS (Ignacio González Fernandez, Guillermo Ibáñez), CNIT (Giuseppe Tropea, Giuseppe Piro, Pietro Boccadoro, Giuseppe Ribezzo, Giulio Rossi, Andrea Detti, Nicola Blefari Melazzi), CRAT (Silvia Canale, Francesco Delli Priscoli, Alessandro Di Giorgio, Federico Lisi, Alessio Martino, Saverio Mascolo, Martina Panfili, Manlio Proia, Lorenzo Ricciardi Celsi, Antonio Pietrabissa, Vittorio Palmisano), FLUIDTIME (Stephan Strodl), SINTEF (Dag Kjenstad, Christian Schulz, Lukas Bach, Carlo Mannino)
Internal Reviewer(s):	CNIT (Andrea Detti)
Abstract:	This document reports the development and testing of the adaptation functionality of the BONVOYAGE mobility platform.
Keyword List:	Adaptation functionality, distribution of spatial data, federated trip planning services, intelligent transportation system, scalability, stress tests, standardized travel data.

TABLE OF CONTENTS

List of Figures

List of Tables

BONVOYAGE Glossary

In Table 1 we have listed and described the terms that have been considered central and relevant in this deliverable.

BONVOYAGE GLOSSARY	
TERM	**DEFINITION**
Adapt	To change something so that it functions better or is better suited for a purpose.
Adaptation Functionality	Functionalities for the external services to be able to communicate with the BONVOYAGE platform.
DATEX II	The standard format for exchange of traffic management information developed in line with the ITS Action Plan.
GTFS	General Transit Feed Specification (GTFS) is an open standard data format for exchange of public transport timetables used by many applications and transport agencies.
Horizontal scaling	To add more nodes into the system such as adding a new computer to a distributed software application.
Integration test	An integration test tests the interaction between different modules of a software project. These tests often have a longer running time and debugging is more difficult and time consuming than unit tests since a failure may depend on specific states in multiple objects.
Metadata Handling Tool	A component of the BONVOYAGE architecture that makes data from external resources available and enriched.
Multi-Modal Mobility Database	A component of the BONVOYAGE architecture that stores all the information necessary for the intelligent functionalities, e.g., user related information and high-level topological information.
NeTEx	NeTEx is the emerging CEN Technical standard for exchanging public transport information as XML documents.
Non-regression test	A non-regression test verifies whether, after introducing or updating a given software application, the change has had the intended effect.
Orchestrator	The BONVOYAGE Orchestrator is a decomposition approach to solve the trip planning on a multimodal network by means of soloists. Orchestrators can act as national access points for trip planning.
Overlay graph	An aggregated, high-level graph used by the BONVOYAGE Orchestrator to organize single soloists into a coherent federation of coordinated trip planning services.
Regression test	A regression test verifies that software which was previously developed and tested still performs the same way after it was changed or interfaced with other

	software.
Scalability	Scalability is the capability of a system to handle a growing amount of work, or its potential to be enlarged to accommodate that growth.
Soloist	The soloists are distributed trip planning services to handle trip planning tasks for a part of the overall transportation and road network.
SPROUTE	SPROUTE is an open source format to exchange routing information (request and response) as JSON objects.
Unit test	A unit test is a test of some (usually minimal) unit. Typically, a unit is a function or a class. Ideally a unit tests does not rely on code in the tested project outside the tested unit.
Vertical scaling	To add resources to a single node in a system, typically involving the addition of CPUs or memory to a single computer.

Table 1: BONVOYAGE Glossary

1 Introduction

1.1 Deliverable Rationale

The goal of this deliverable is to provide an implementation and test report, for the components providing the adaptation functionality. Therefore, details about the implemented software and the results of the performance tests are key focus areas.

The deliverable builds on top of several previous deliverables which therefore are frequently referenced:

- D2.2: BONVOYAGE Architecture
- D4.1: Design of the Intelligent Transport Functionality
- D4.2: Development and validation of the Intelligent Transport Functionality
- D5.1: Design of the adaptation functionality
- D6.1: Technology dependent interfaces
- D6.2: Apps
- D7.1: Integration plan

We define *Adaptation Functionality* primarily as the functionalities for the external services to be able to communicate with the BONVOYAGE platform. The realization of these functionalities involves several components, most noticeably:

- The Multi-Modal Mobility Database which enables storage of BONVOYAGE data
- The Metadata Handling Tool which enables access to externally available data
- The Trip Planning Services which adapts trip planning algorithms into services compliant with the BONVOYAGE communication protocols.

The implementation and testing of these components are found in this deliverable while the behaviour of the fully integrated system will be described in a later document, D7.2: System Integration Report.

1.2 Quality review

VERSION CONTROL TABLE			
VERSION NO.	**PURPOSE/CHANGES**	**AUTHOR**	**DATE**
0.1	FRONT PAGE AND TENTATIVE TOC	DAG KJENSTAD	01/08/2017
0.2	DISTRIBUTION OF CONTRIBUTIONS	DAG KJENSTAD	29/08/2017
0.3	ADDED CONTENTS ABOUT TRIP PLANNING SERVICES	DAG KJENSTAD	08/09/2017
0.4	ADDED CONTRIBUTIONS FROM CRAT	FEDERICO LISI	10/09/2017
0.5	ADDITIONAL ORCHESTRATOR DOCUMENTATION	DAG KJENSTAD	11/09/2017
0.6	ADDED CONTRIBUTIONS FROM ATOS	GUILLERMO IBÁÑEZ	11/09/2017
0.7	ADDED CONTRIBUTIONS FROM FLUIDTIME	STEPHAN STRODL	12/09/2017
0.8	ADDED CONTRIBUTIONS FROM CNIT	GIUSEPPE TROPEA	12/09/2017
0.9	COMPLETE VERSION FOR INTERNAL REVIEW	GIUSEPPE TROPEA	14/09/2017
1.0	FINAL VERSION	DAG KJENSTAD, ANDREA DETTI, GIUSEPPE TROPEA	15/09/2017

Table 2: Document versions

Please notice that for this deliverable the contractual date has been officially moved to September 15, 2017.

1.3 Executive summary

1.3.1 Deliverable description

The Multi-Modal Mobility Database (MMMDB) is a core component of the adaptation functionality. The data model was already described in D5.1. In section 2.1 we briefly describe the current database implementation before we go in more depth regarding its use, workflow and error handling while storing and retrieving user profiling data.

Section 2.2 is dedicated to the implementation details regarding the Metadata Handling Tool (MDHT), a key component for external services to be able to communicate with the BONVOYAGE platform. The MDHT accesses external data sources to obtain the relevant travel data, analyses and enriches the data with metadata important for BONVOYAGE other components, and makes it available for those components. We describe the tests of the MDHT in section 3.1.

The design of the BONVOYAGE architecture has evolved towards being a potential continent-wide federated platform of collaborative but independent trip planning services and data-sources. Hence, the implementation of individual components must allow for scalability, i.e., horizontal scaling by adding more nodes into the system or vertical scaling by adding more computational resources to single nodes, which is described in section 2.3. We have therefore

performed tests on the trip planning components to see if this is possible and gives the desired effect. The test setup and results are found in section 3.2.

In addition to this document the deliverable also consists of the actual software used. In section 4 we have listed the components and where the software source code and database extracts can be found.

1.3.2 Summary of results

The results of the work carried out within the scope of WP5 can be summarized as follows:

- We have made a comprehensive design of the adaptation functionality in accordance with the design from WP2. The design provides as a continent-wide federated platform of collaborative but independent travel planning services and data-sources, and is documented in D5.1.
- We have implemented the core components of the adaptation functionality which are ready for the ongoing system integration and validation of WP7.
- We have tested these components with unit tests and integration tests. Some regression tests are added to detect undesired changes in behaviour as updates of individual components are made.
- We have tested the performance of the Metadata Handling Tool. The tests show a noticeable performance increase by developing a car-sharing MDHT adaptor. Further details for this will be given in upcoming D7.x deliverables where the integration is described in details.
- We have tested the trip planning services for scalability. Exact trip planning algorithms have exponential complexity, and hence the trip planning services can easily become a bottleneck resource in the overall trip planning architecture. Careful implementation and testing is therefore necessary. The results of the load stress performance tests show that an implementation of orchestrators and soloists as multiprocessing services can provide fast response times also when the number of trip requests is large. We also believe that the approach is applicable for horizontal scaling.

2 Implementation of components

2.1 Implementation of the Multi-Modal Mobility Database

In the following paragraphs the reader can find implementation details about how the MMMDB is used by the various pieces of the BONVOYAGE platform. It specifically impacts the user profiling capabilities and the interfacing of the components with the Application Server.

2.1.1 Deployment and availability

This database was initially designed and implemented (as a stand-alone component) using the MySQL DBMS, following the Entity Relationship diagram which can be seen in Figure 57 of Appendix A – User Data Model, in Deliverable 4.1. Several partners started by deploying their own independent copies of the MMMDB. Later on, we have changed in favour of a spatial database system, because implementation, testing and debug activities of BONVOYAGE components are much easier if the MMMDB natively supports spatial operations. It was then agreed to deploy the MMMDB on a PostgreSQL server, so that the partners could exploit the PostGIS extension, available in this DBMS.

Hence a migration step was performed, switching the schema from MySQL to PostrgreSQL.

At the same time a central MMMDB was hosted on a machine at ATOS to make it available for the whole team, and to allow its shared use for implementation and testing of several components of the platform. However, following the internal rules of the company, the hosting machine and DBMS were accessible only using an SSH tunnel, and that was an issue for certain activities.

It was decided to move the MMMDB to publicly available server in Roma, managed by CNIT. Therefore, a dump of MMMDB was done and sent to CNIT to ease its installation. A snapshot of it is reported In Appendix A for reference purposes.

Figure 1 shows an active connection to MMMDB, using PGAdmin4, and some of its tables.

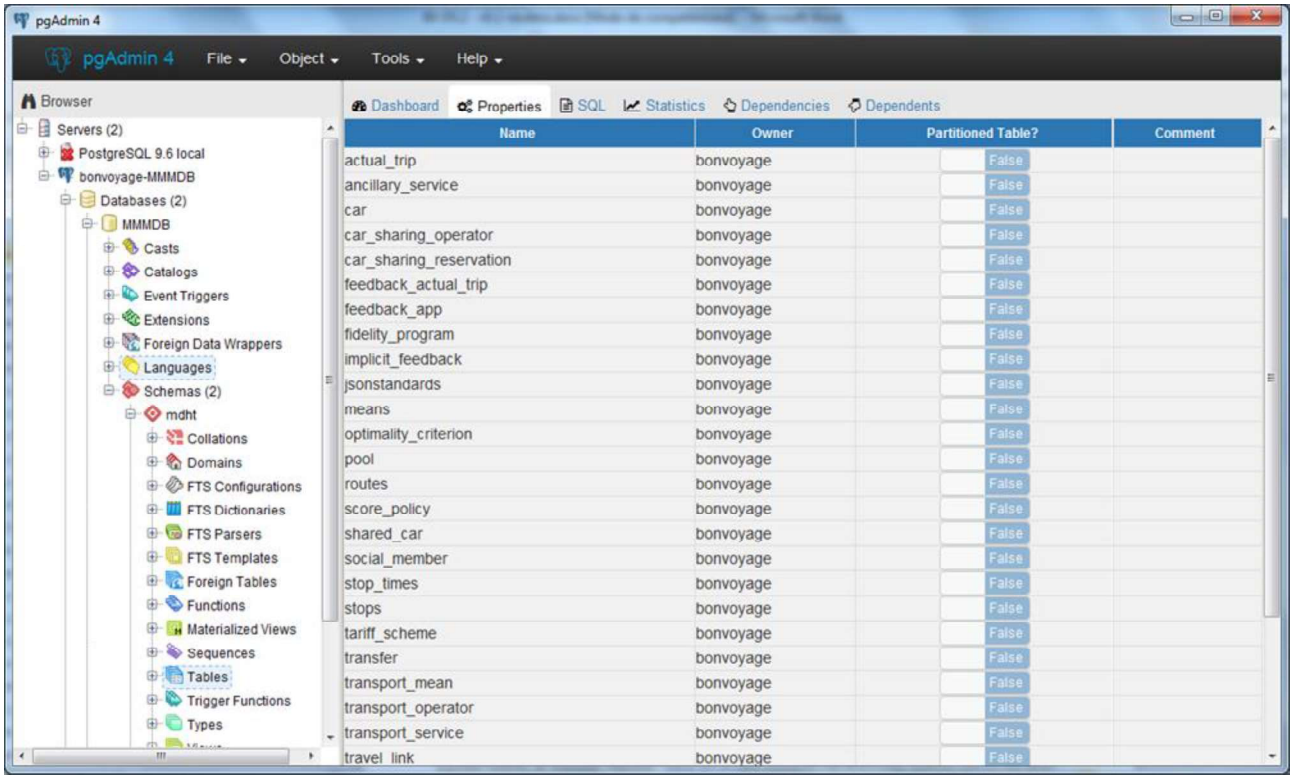

Figure 1: Connection to MMMDB via PGAdmin4, showing some of its tables.

2.1.2 Interfacing with the Application Server

The MMMDB is used as data storage for all Application Server modules. The User Profiling, Feedback, Selected Routes and Greenpoints data are stored in and received from the MMMDB. The Application Server tests the connection and the right configuration of the storage at start-up. The initial setup of the Application Server triggers a test routine to connect to the database. After successful execution of the test the application service is available otherwise error reports are generated in the Application Server's log files.

The connection might fail during operation. In this case the Application Server will keep trying to reconnect to the database. This auto connect/auto-retry capability improves exception handling when the MMMDB is temporarily down. In other cases, error handling and reporting ensures appropriate treatment of the lost or unsuccessful connection. Initial testing of the MMMDB was conducted by writing and reading known values to the data store using the connection details, provided credentials and sensible system settings for the high performance HikariCP JDBC connection pool used.

As an example, User Profiling is in charge of computing (i) a pattern for each user on the basis of the current user query and (ii) a ranking of the list of travel solutions given back to the user on the basis of her individual behavioural context (acquired via explicit feedback such as the selected travel solution). The services involved in the online pattern identification and online behavioural context acquisition are *Online User Profiling* and *Rank Tool*. For what

concerns *Online User Profiling*, we use data instances stored in the *User_Query, Registered_User* and *User_Class* tables of the Multi-Modal Mobility Database for identifying the pattern in order to provide the optimality criteria for personalizing the travel solutions. Similarly, we use data instances stored in the *User_Profile* table, thus taking information from each traveller's behavioural context, in order to provide them a set of ranked travel solutions.

Please notice that these services have been widely described from the methodological and algorithmic point of view in Deliverable D4.1, whereas test and validation results took place in Deliverable D4.2. We refer to these documents (D4.1 and D4.2, both already submitted at the time of this writing) for a complete description and all references.

2.2 Implementation of the Metadata Handling Tool

2.2.1 What is Metadata Handling Tool?

MDHT is part of the Infrastructure Layer, and **runs as data producer**, producing information that feeds the Discovery and Pub/Sub functionalities of the platform.

Its main function can be defined as **to periodically access transport operators' data sources, obtain travel data and services information, analyse it, extract metadata items, and insert them into** two other components that work as sources of refined and aggregated information within the BONVOYAGE platform: OpenGeoBase and Internames Communication System's Publish/Subscribe Services.

So, the components of BONVOYAGE platform that are directly related with MDHT are (see Figure 2):

- **Data Sources of Transport Operator**. These act as data source for the BONVOYAGE platform. MDHT's main job is to obtain meta-information from them, performing adaptation to the different technologies and formats. These external data sources and services are scanned and spatially indexed for discovery purposes and data coming from them is federated (so that the platform knows how to access it) or proxied, in order to facilitate data acquisition.
- **OpenGeoBase (OGB)**. This acts as a destination for MDHT, and as a source of metadata for other platform components. MDHT must insert metadata into OGB in terms of coordinate pairs and contact information, extracted from the original data from the transport operators, which is vital for the purpose of BONVOYAGE platform components to be aware of what data is available outside of the platform.
- **Internames Communication System Publish/Subscribe Service (ICS PUB/SUB)**. This acts as a data destination for MDHT, but as a data source for other platform components. MDHT must send real-time and incrementally changing data to it, extracted from transport operators' data, for the purpose of efficiently offering information that would cost a severe performance hit if accessed through the original service.

Please refer to the following documents for a comprehensive understanding of the design and purpose of the MDHT:

- D5.1 paragraph 3.2, for the concept of data discovery and the design of the GeoJSON for ITS format able to capture relevant metadata about travel data sources, and how OGB supports that concept. Also, see paragraph 2.2.8 of this deliverable for the detailed design of the GeoJSON for ITS format.

- D5.1 chapter 5 and D5.2 (this deliverable) at paragraph 2.2.9, for the concept of intercepting streams of real-time data, parsing them and feeding them to pub/sub channels of the Internames Communication System, in order to guarantee efficient access to real-time data sources of information about travel and transit.
- D7.1 paragraph 2.1.10, for the plan of an integrated design of the MDHT within the platform.

The following sections of this deliverable, instead, bring focus to the detailed implementation of the modular architecture of the MDHT itself.

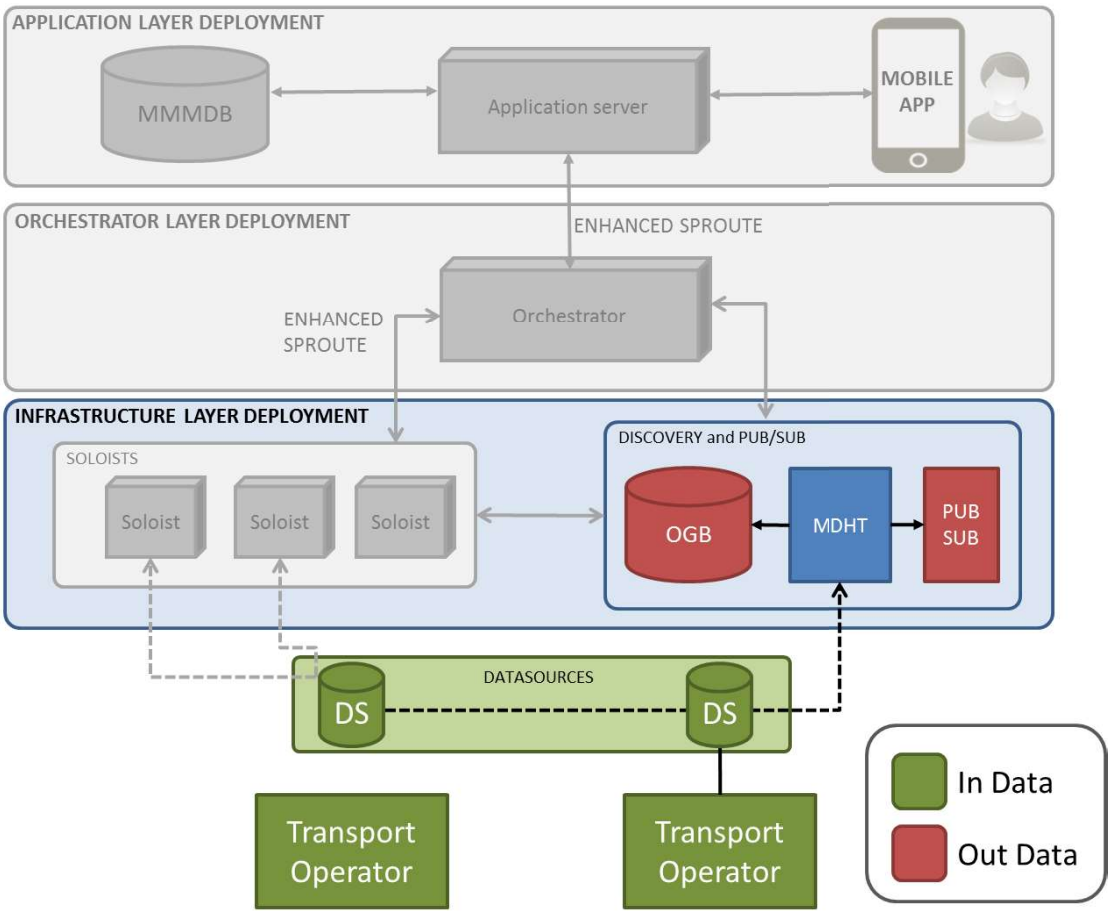

Figure 2: Components directly related with MDHT and how they work.

2.2.2 What Metadata Handling Tool is not

It's important to define what the MDHT is not, in terms or what it doesn't do at its interface, to provide a better understanding of its implementation.

- It *isn't a data source*. It's a data source scanner and metadata producer that feeds information to other data distributor components in BONVOYAGE.

- This means that *other components don't directly request data* from it. In other words, *it doesn't provide on-demand services* to other components.
- It isn't dependant on other components. Inside the BONVOYAGE platform, *it works as an independent thread*.

2.2.3 Specific objectives

- **OBJ-01 Monitor transport operator data sources.** Review transport operator data sources periodically, looking for new data or change in them. Critical to this goal, since the whole list of transport operators and their data sources is ever changing, is the modular architecture and implementation, based on technology-specific adaptors.
- **OBJ-02 Fetch data files from Transport Operator Data Source.** Study different ways to access and get the data files from Data Sources of the different Transport Operators, and implement it.
- **OBJ-03 Create adaptors for each specific Data Source technology.** Create an adaptor for each technology and end points where travel data will be fetched from, to standardize the procedure as much as possible. The list of supported technologies so far is: *NeTEx*, *DATEX II*, *GFTS files*, the *Bilbao CoCities* platform, the *Norwegian NPRA* server, *car2go, SHARE'Ngo and Enjoy* car sharing operators.
- **OBJ-04 Create metadata for OGB.** Analyse and extract meta-information, fetching from Transport Operators' original sources, that has to be inserted into OGB. The information is to be mapped into the GeoJSON for ITS format.
- **OBJ-05 Send adapted data to ICS PUB/SUB Service.** Parse, re-package and convert data, fetching from Transport Operators' original sources, which have to be sent to ICS PUB/SUB Service or to other re-distribution and caching sub-systems.
- **OBJ-06 Manage the component.** Provide functions and APIs that allow a user (it can be BONVOYAGE platform or an external human user) to manage the MDHT component: start, stop and consult data and statistics.

2.2.4 Functional requirements

2.2.4.1 Defining actors

In this section the actors who interact with the MDHT are described.

- **ACT-01 BONVOYAGE system**. This actor is the **generic and main user** of the MDHT application. As the application runs automatically (although it interacts with other BONVOYAGE components to fetch or insert data into them), it doesn't receive any interactive directives from the outside and it can be considered and independently running thread. However, the global BONVOYAGE system is the responsible for starting the various MDHT operations.
- **ACT-02 External user**. This actor is an **external human user**, who is any user that can also administer or manage globally the BONVOYAGE platform. This user is the external user who has the rights to start, stop and manage the global BONVOYAGE system. He/she can also manually start, stop and manage the MDHT component alone.
- **ACT-03 Scheduler**. This actor is a representation of an **internal user**, which is a system process responsible for launching the main tasks. This user is the responsible for launching processes of collection, analysis, extraction and insertion of data.

- **ACT-04 Transport Operator Data Source**. This represents a **set of components** that act as data providers. This user acts as data provider from Transport Operators (TO). It doesn't interact with system; it just provides information whenever it's required.
- **ACT-05 OpenGeoBase**. This is a **component within BONVOYAGE**. This user acts as data consumer. It doesn't actively modify the behaviour of MDHT; it just receives specific information whenever MDHT produces it.
- **ACT-06 ICS PUB/SUB Service**. This is a **component within BONVOYAGE**. This user acts as data consumer. It doesn't actively modify the behaviour of MDHT; it just receives specific information whenever MDHT produces it.

2.2.4.2 Use case diagrams

The expected functionalities of the developed software are represented graphically in the following. The requirements can be divided into two groups ("Scheduler control" and "Data processing") according to their functionality. Figure 3 and Figure 4 graphs shows the division of the two different initial subsystems.

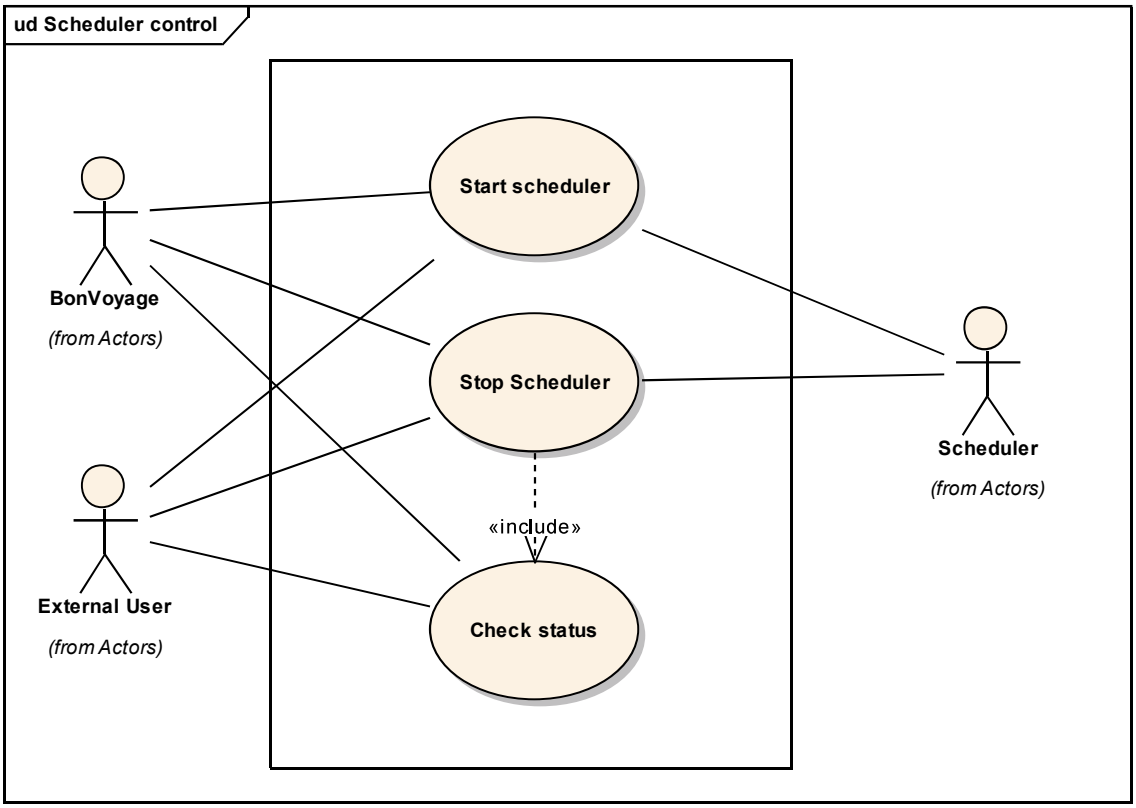

Figure 3: Scheduler control use case diagram.

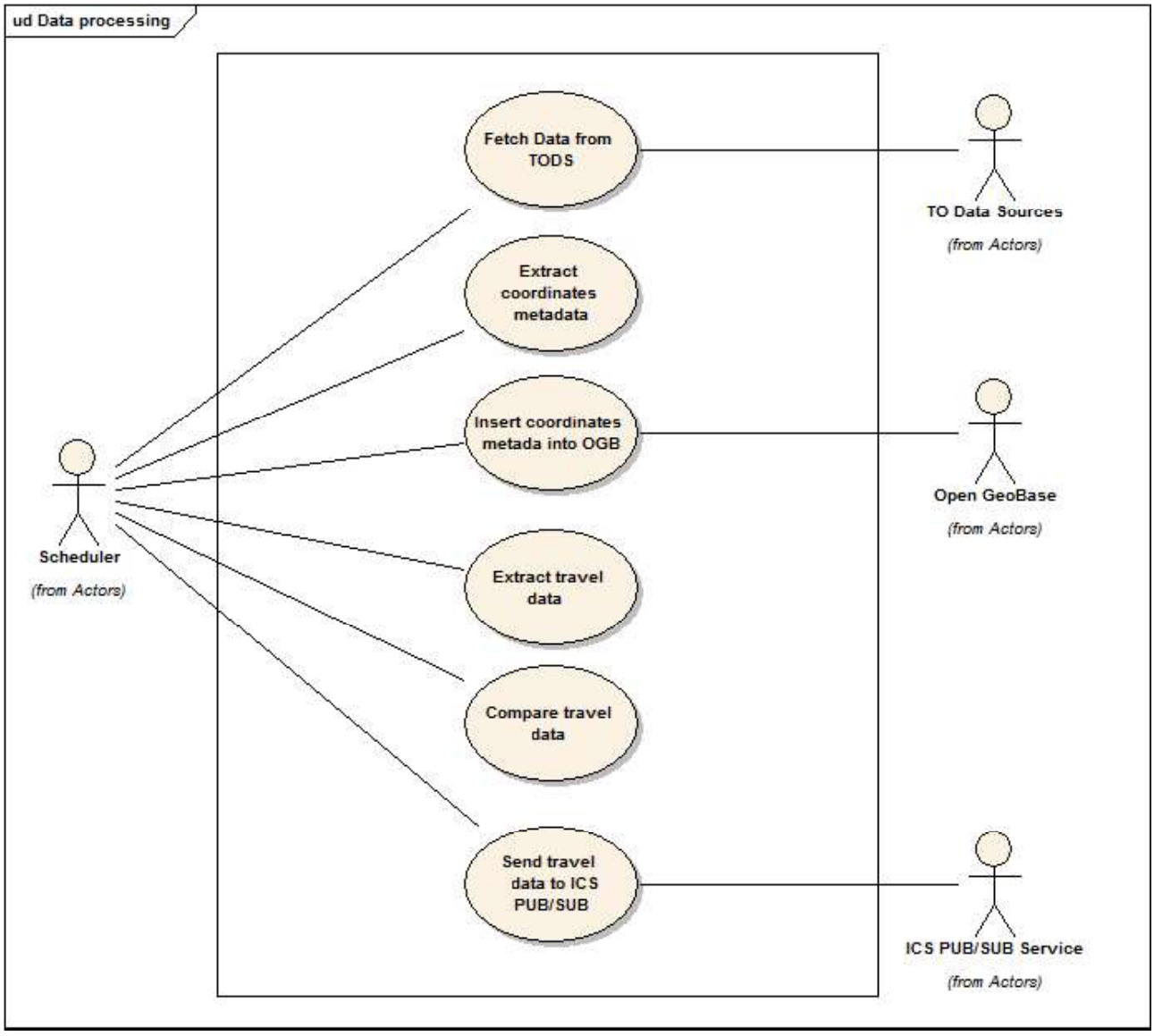

Figure 4: Data processing use case diagram.

2.2.4.3 System use cases

2.2.4.3.1 Scheduler control subsystem use cases

FR-01	Start scheduler
Related objectives	OBJ-06. Manage the component.
Description	The system should behave as described in the following use case, when *the user wishes to start the application*.

Precondition	To have access to BONVOYAGE platform with administration permissions.
Normal sequence	**Step** **Action** 1 The **BONVOYAGE system actor (ACT-01)** or **External user actor (ACT-02)** request the system to *start the task scheduler*. 2 The system launches the scheduler start sequence: first, getting the **list of adaptors Transport Operators Data Sources and their information from an internal database and loading it in internal array**. The information related with each Data Source must be, at least: • **Name of Transport Operator** or its Data Source. • **URL**, from the data will be downloaded. • **Period**, how long (in seconds) it takes to repeat the download and subsequent operations. 3 The system **launches an action thread** for each item of array: **download data**, **analyse** it**, extract metadata** and **insert it into OGB, and publish into ICS PUB/SUB Service,** if needed.
Postcondition	
Exceptions	**Step** **Action** 2 If the list of adaptors of Transport Operators Data Source is empty or inaccessible, an exception occurs that returns a message reporting the error.
Performance	**Step** **Time frame** 2 1 second 3 5 seconds
Frequency	Once a day
Priority	High
Comments	In case of the *BONVOYAGE system actor*, access to the start-up function will occur through an API. In case of the *External user actor*, access to the start-up function will occur through a web service.

Table 3:* Start scheduler *use case specification.

FR-02	Stop scheduler
Related objectives	OBJ-06. Manage the component.
Description	The system should behave as described in the following use case, when *the user wishes to stop the application*.
Precondition	To have access to BONVOYAGE platform with administration permissions.
Normal sequence	**Step** **Action** 1 The **BONVOYAGE system actor (ACT-01)** or **External user actor (ACT-02)** request the system to *stop the task scheduler*. 2 The system launches the scheduler stop sequence: first, **checking the system status**, including each of thread launched. 3 **When one thread finishes, it is stopped**. And so on, until all threads are stopped.
Postcondition	
Exceptions	**Step** **Action** - -
Performance	**Step** **Time frame** 2 1 second 3 10 seconds – 60 seconds. Depending on the number of threads launched and their status when stop command is received.
Frequency	Once a day
Priority	High
Comments	In case of the *BONVOYAGE system actor*, access to the stop function will occur through an API. In case of the *External user actor*, access to the stop function will occur through a web service.

Table 4:* Stop scheduler *use case specification.

FR-03	Check status
Related objectives	OBJ-06. Manage the component.
Description	The system should behave as described in the following use case, when *the*

	user wishes to check the status of the application.
Precondition	To have access to BONVOYAGE platform with administration permissions.
Normal sequence	**Step** **Action** 1 The **BONVOYAGE system actor (ACT-01)** or **External user actor (ACT-02)** request the system to *check the status of the task scheduler*. 2 The system **captures the status of each thread** launched. 3 The system sends, or shows, a report with the status of each thread, that can be: • **Empty**: it's in the interval between one action and the next. • **Fetching data**: it's fetching data from Transport Operator Data Source. • **Analysing data**: it's analysing data. • **Extracting metadata OGB**: it's extracting metadata for OGB. • **Extracting metadata ICS**: it's extracting metadata for OCS PUB/SUB Service. • **Inserting into OGB**: it's inserting metadata into OGB. • **Sending to ICS**: it's sending metadata to ICS PUB/SUB Service.
Postcondition	
Exceptions	**Step** **Action** - -
Performance	**Step** **Time frame** 2 1 second 3 1 second – 5 seconds. Depending on the number of threads launched and their status when stop command is received.
Frequency	Once a day
Priority	High
Comments	In case of the *BONVOYAGE system actor*, the report of status is through an API. In case of the *External user actor*, the report is showed through a web service.

Table 5:* Check status *use case specification.

2.2.4.3.2 Data processing subsystem use cases

FR-01	Fetch data from TODS
Related objectives	OBJ-01. Monitor transport operator data sources. OBJ-02. Fetch data files from Transport Operator Data Source.
Description	The system should behave as described in the following use case, when *the scheduler (system) starts the sequence to fetch data from TODS (Transport Operators Data Sources)*.
Precondition	The Transport Operators Data Sources list and its information must have been previously loaded in the system.
Normal sequence	**Step** **Action** 1 The **Scheduler actor (ACT-03)** requests the system to *fetch data from a Transport Operator Data Source*. 2 The system **tries to access and download the file or data from TO Data Sources (ACT-04)**. 3 The file or data is stored in the system. 4 When the whole file or data has been downloaded to the system, it will be noted in a *log file*.
Postcondition	The files (or data) must be saved in the system, inside a folder with the name of the Transport Operator, renaming it, adding to its name the complete date (*date+time*).
Exceptions	**Step** **Action** 2 If the system can't download the file or data from Data Source, it will be noted in the log file, and the system will wait until the established period.
Performance	**Step** **Time frame** 2 10 seconds – 60 seconds, depending on the file or data size and the connection speed of the network.
Frequency	Depending on the period established in the data source information.
Priority	High
Comments	-

Table 6:* Fetch data from TODS *use case specification.

FR-02	**Extract coordinates and other metadata**
Related objectives	OBJ-03. Create adaptors for each specific Data Source technology. OBJ-04. Create metadata for OGB.
Description	The system should behave as described in the following use case, when *the scheduler (system) starts the sequence to analyse and extract coordinates and other metadata* from the file or content downloaded from the Transport Operator Data Source.
Precondition	The previous step (FR-04) must have finished successfully.
Normal sequence	**Step** **Action** 1 The **Scheduler actor (ACT-01)** requests the system to *extract coordinates and other metadata*. 2 The system locates and gets the file or content downloaded from the transport operator. 3 The system analyses the content, using the correct patch according to the file format. 4 The system locates and extracts coordinate pairs, creating an *ArrayList<double[]>* item with them, and setting result flag to *SUCCESS*. 5 When the whole file or data has been analysed, the result summary is noted in the *log file*.
Postcondition	The result flag of the process (setting to *SUCCESS*), and the *ArrayList<double[]>* item, have to be returned to **Scheduler actor (ACT-01)**.
Exceptions	**Step** **Action** 3 If the analysing process can't process the file, it's stopped and noted in the *log file*, and the result flag is set to *FAILURE_BY_FORMAT*, indicating that the result has been negative because of the format of the content. 4 If the file doesn't contain coordinate pairs, the *ArrayList<double[]>* item isn't created, and the result is set to *NO_DATA*, indicating that the result is empty, because the content doesn't contain coordinate pairs. This information is noted in the *log file* too.
Performance	**Step** **Time frame** 3 1 seconds – 10 seconds, depending on the file or data size.

Frequency	Depending on the period established in the data source information.
Priority	High
Comments	-

Table 7: **Extract coordinates and other metadata *use case specification.***

FR-03	**Insert coordinates and other metadata into OGB**
Related objectives	OBJ-04. Create metadata for OGB.
Description	The system should behave as described in the following use case, when *the scheduler (system) starts the sequence to insert coordinates and other metadata into OGB*.
Precondition	The previous step (FR-05) must have obtained coordinate pairs.
Normal sequence	**Step** **Action** 1 The **Scheduler actor (ACT-01)** requests the system to *insert coordinates and other metadata into OGB*. 2 The system opens connection with **OpenGeoBase (ACT-05)**. 3 The system uses the **OpenGeoBase (ACT-05)** API function to insert coordinate pairs (located in *ArrayList<double[]>* item) into it. 4 The **OpenGeoBase (ACT-05)** returns the result of the process, with coordinate pairs in JSON format, and the id of the created item into it. 5 The data returned by **OpenGeoBase (ACT-05)** is noted in the *log file*, and the result flag is set to *SUCCESS*.
Postcondition	-
Exceptions	**Step** **Action** 2 If the system can't connect to the OGB, it will be noted in the *log file*, and the result flag is set to *NO_CONNECTION*, indicating that it hasn't been possible to insert coordinates and other metadata into OGB because there is no connection.
Performance	**Step** **Time frame**

	2 3 seconds – 10 seconds, depending on the connection to OGB.
Frequency	Depending on the period established in the data source information.
Priority	High
Comments	-

Table 8: **Insert coordinates and other metadata into OGB *use case specification.***

FR-04	**Extract real-time travel data**
Related objectives	OBJ-03. Create adaptors for each specific Data Source technology. OBJ-05. Send adapted data to ICS PUB/SUB Service.
Description	The system should behave as described in the following use case, when *the scheduler (system) starts the sequence to analyse and extract real-time travel data* from the file or content downloaded from the Transport Operator Data Source.
Precondition	The previous step (FR-04) must have finished successfully.
Normal sequence	**Step** **Action** 1 The **Scheduler actor (ACT-01)** requests the system to *extract real-time travel data*. 2 The system locates and gets the file or content downloaded from the transport operator. 3 The system analyses the content, using the correct patch according to the file format. 4 The system locates and extracts *real-time travel data*, creating a *JSON element* with them, and setting result flag to *SUCCESS*. 5 When the whole file or data has been analysed, the result summary is noted in the *log file*.
Postcondition	The result flag of the process (setting to *SUCCESS*), and the *JSON element*, have to be returned to **Scheduler actor (ACT-01)**.
Exceptions	**Step** **Action** 3 If the analysing process can't process the file, it's stopped and noted in the *log file*, and the result flag is set to *FAILURE_BY_FORMAT*, indicating that the result has been negative because of the format of the content.

	4	If the file doesn't content real-time travel data, the *JSON element* isn't created, and the result is set to *NO_DATA*, indicating that the result is empty, because the content doesn't contain coordinate pairs. This information is noted in the *log file* too.
Performance	**Step**	**Time frame**
	2	1 seconds – 10 seconds, depending on the file or data size.
Frequency	Depending on the period established in the data source information.	
Priority	High	
Comments	-	

Table 9: **Extract real-time travel data *use case specification.***

FR-05	**Compare real-time travel data**	
Related objectives	OBJ-05. Send adapted data to ICS PUB/SUB Service.	
Description	The system should behave as described in the following use case, when *the scheduler (system) starts the sequence to compare the obtained real-time travel data with those obtained and stored during the previous processing*.	
Precondition	The previous step (FR-07) must have obtained real-time travel data.	
Normal sequence	**Step**	**Action**
	1	The **Scheduler actor (ACT-01)** requests the system to *compare real-time travel data obtained currently with real-time travel data obtained previously, and stored in an internal DB*.
	2	The system retrieves from an internal DB the real-time travel data obtained the last time.
	3	The system compares both of real-time travel data, and if there are differences the result flag is set to *DIFFERENT*.
	4	When the comparing process has finished, the result will be noted in a *log file*.
Postcondition	The result flag of the process (setting to *DIFFERENT*) must be returned to **Scheduler actor (ACT-01)**.	
Exceptions	**Step**	**Action**
	2	If the system can't locate in the internal DDBB the real-time travel data obtained in previous time, the result flag is directly set to

	DIFFERENT.
	3 If there is no difference between them, the result flag is set to *NOT_DIFFERENT*, returning this result to **Scheduler actor (ACT-01)**, and noting it in *log file*.
Performance	**Step** **Time frame**
	3 1 seconds – 5 seconds, depending on the real-time travel data to compare.
Frequency	Depending on the period established in the data source information.
Priority	High
Comments	-

Table 10: **Compare real-time travel data *use case specification.***

FR-06	**Send real-time travel data to ICS PUB/SUB**
Related objectives	OBJ-05. Send adapted data to ICS PUB/SUB Service.
Description	The system should behave as described in the following use case, when *the scheduler (system) starts the sequence to send real-time travel data to ICS PUB/SUB Service*.
Precondition	The previous step (FR-08) must have obtained different real-time travel data from those obtained during the last processing.
Normal sequence	**Step** **Action**
	1 The **Scheduler actor (ACT-01)** requests the system to *send real-time travel data to ICS PUB/SUB Service*.
	2 The system opens connection with **ICS PUB/SUB Service (ACT-06)**.
	3 The system uses the **ICS PUB/SUB Service (ACT-06)** API function to insert *real-time travel data*, via *JSON element*.
	4 The **ICS PUB/SUB Service (ACT-06)** returns the result of the process.
	5 The data returned by **ICS PUB/SUB Service (ACT-06)** is noted in the *log file*, and the result flag is set to *SUCCESS*.
	6 Once the process has been successful, the new data sent to **ICS PUB/SUB Service (ACT-06)** is inserted in the internal DB.

Postcondition	-	
Exceptions	**Step**	**Action**
	2	If the system can't connect to ICS Service, it will be noted in the *log file*, and the result flag is set to *NO_CONNECTION*, indicating that it hasn't been possible to insert coordinates and other metadata into OGB because there is no connection.
Performance	**Step**	**Time frame**
	2	3 seconds – 10 seconds, depending on the connection to ICS Service.
Frequency	Depending on the period established in the data source information.	
Priority	High	
Comments	-	

Table 11: **Send real-time travel data to ICS PUB/SUB** ***use case specification.***

2.2.4.4 Technological requirements

As this MDHT application is part of a bigger platform, the technological requirements are influenced by those of the bigger system.

In any case, the application will run in a **server** (with any OS like Windows or Linux), with enough free space to host the **application server system** (**Apache-Tomcat**), **DDBB server**, **backend server**, and all information produced by the system and applications within it.

2.2.4.5 Design requirements

NFR-01	**Adaption to parent system**
Related objectives	-
Description	The application must be developed following the design criteria used for the larger system. So, it has to **use same resource**s as it, like server, and it should **use same technology, like Java, JSON, PostgreSQL**, etc. to ease the integration.
Comments	-

Table 12: **Adaption to parent system** ***non-functional requirement description.***

NFR-02	**Multithreading process**

Related objectives	-
Description	Although the process is the same for all information providers, and there will be several resources shared between them, like OGB or ICS Service, the application should manage data from different kind of sources, with different formats, different intervals to update the data, etc. So, it is necessary to use **multithreading programming**, with **one thread per source** and **synchronous access to sharing resources**, like PostgreSQL DB, OGB or ICS Service.
Comments	-

Table 13: **Multithreading process** ***non-functional requirement description.***

NFR-03	**Maintenance**
Related objectives	-
Description	Although the system can grow in the future, with multiple new adapters, as of now it is only contemplated the management of a limited number of sources, previously agreed.
Comments	-

Table 14: **Maintenance** ***non-functional requirement description.***

2.2.5 Class diagram

In Figure 5, it is represented a draft design of a possible classes diagram, and relation between them, that can implement the solution for the above described use cases.

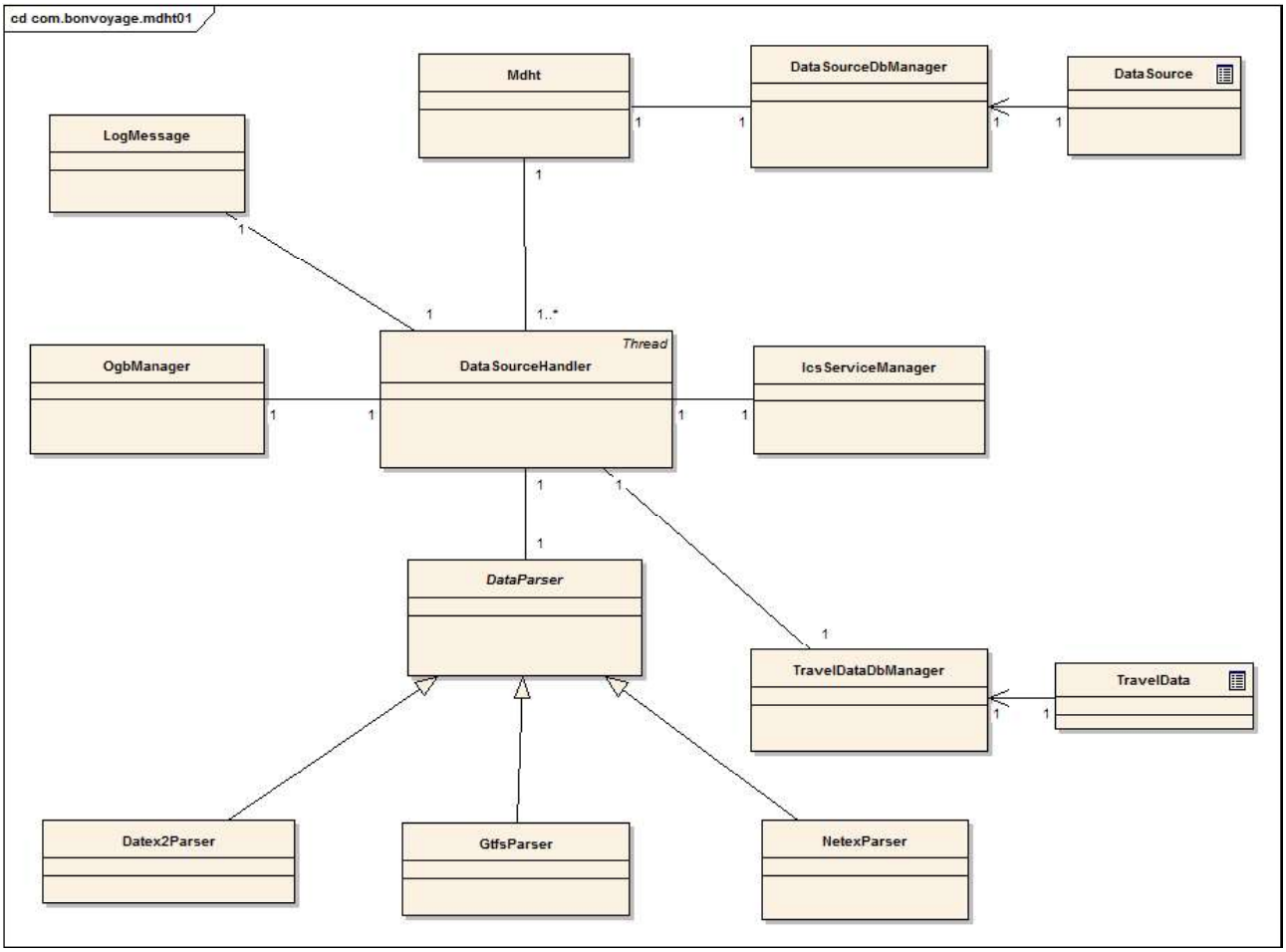

Figure 5: Metadata Handling Tool class diagram.

2.2.6 Sequence diagrams

In the following paragraphs and graphs, we can see the behaviour of the system through sequence diagrams, which show the function of each object and the interactions between them. For a clearer display, it has been decided to show the steps of the behaviour of the objects and their interactions by functionality groups as follows:

- Start global process.
- Processing data source.
- Check status.
- Stop global process.

2.2.6.1 Start global process

The diagram in Figure 6 shows how the initial process runs. Basically, the system receives the order to start, and it gets all information about Transport Operator Data Sources, from the internal database, and launches one thread for each Transport Operator Data Source.

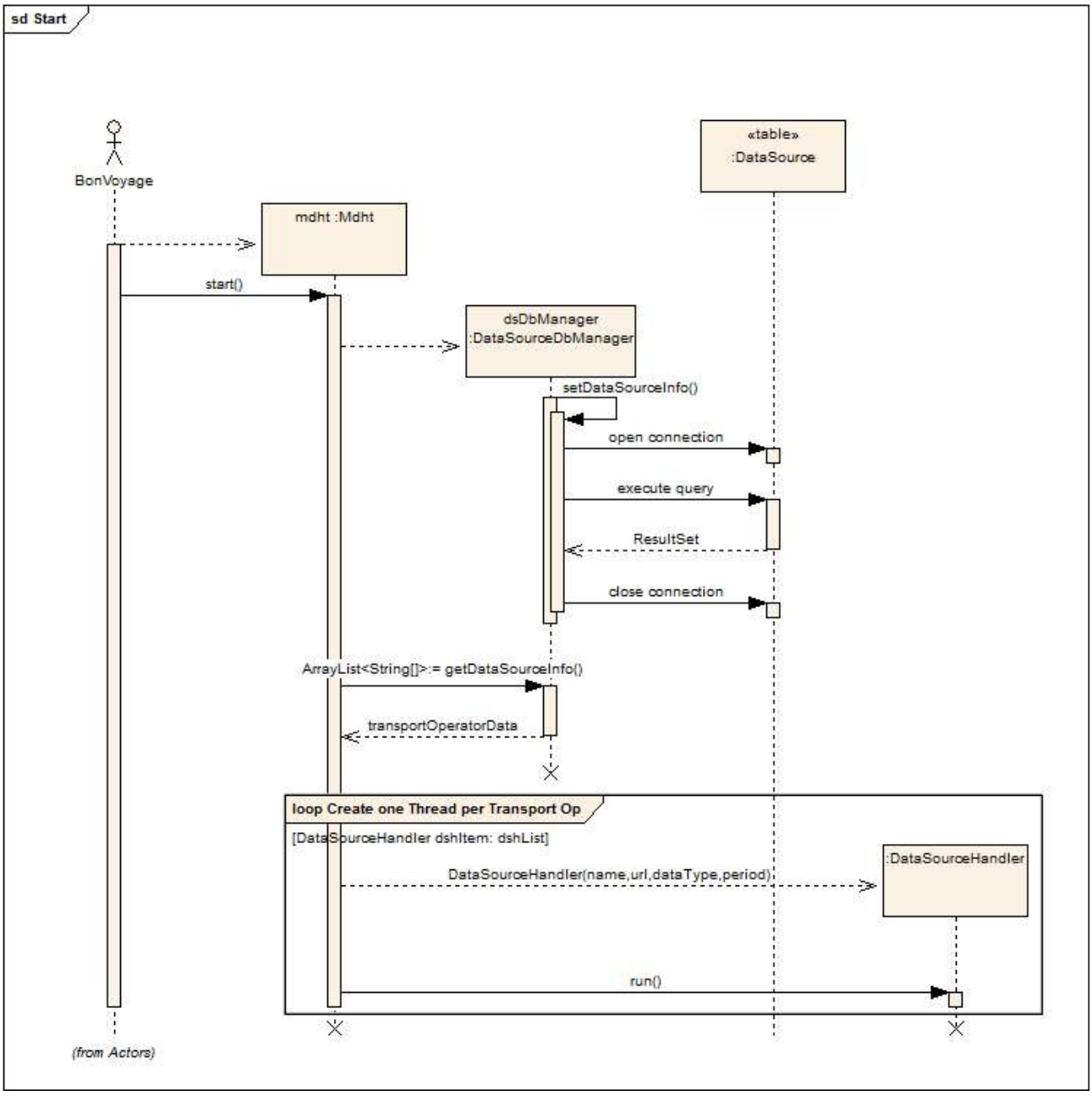

Figure 6: MDHT Start process sequence diagram.

2.2.6.2 Processing data source

Error! Reference source not found. shows the sequence of the processing of a data source of a transport operator. This is something that repeats itself for all of the transport operators.

The sequence, basically, is as follows:

- First, the class that is responsible of processing a data source (Data Source Handler) fetches data from the Transport Operator URL.
- The data is stored in a local drive as local file, in a special folder called source, within the folder of that Transport Operator.
- Then the file is validated according to the format to which it corresponds.
- The next 2 steps are to extract metadata and coordinate pairs and insert them into the OpenGeoBase, using its API.
- The next 3 steps are to extract real-time travel data, compare them with the last ones extracted previously, and, if they are different, send it to ICS Pub/Sub Service and save it a local database, to ease the comparing process.
- The last step puts the process on hold, until the time of the period is over. Then the process will be restarted.

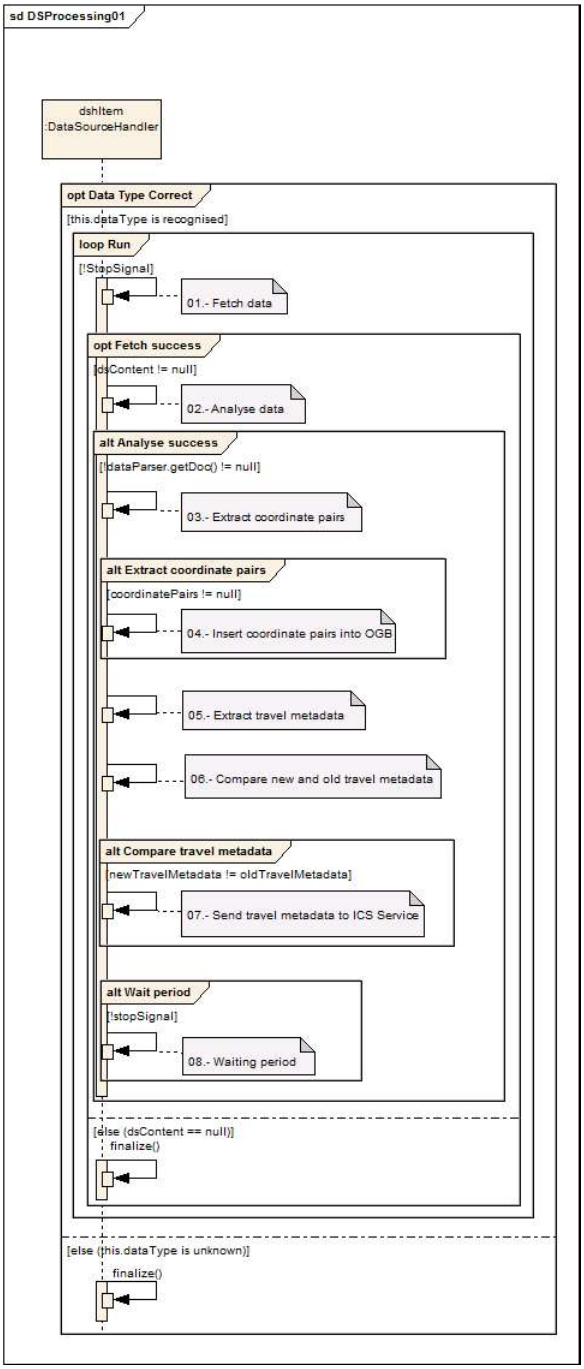

Figure 7: MDHT Data Source Processing sequence diagram.

2.2.6.3 Check status

Here, the process of status checking is showed. The main class manages the process, started by an external user, i.e. the BONVOYAGE system or a user with administrator permissions, who asks for the status of all Data Source Handler instances. When all responses are received, the information is merged in one message that is sent to the user.

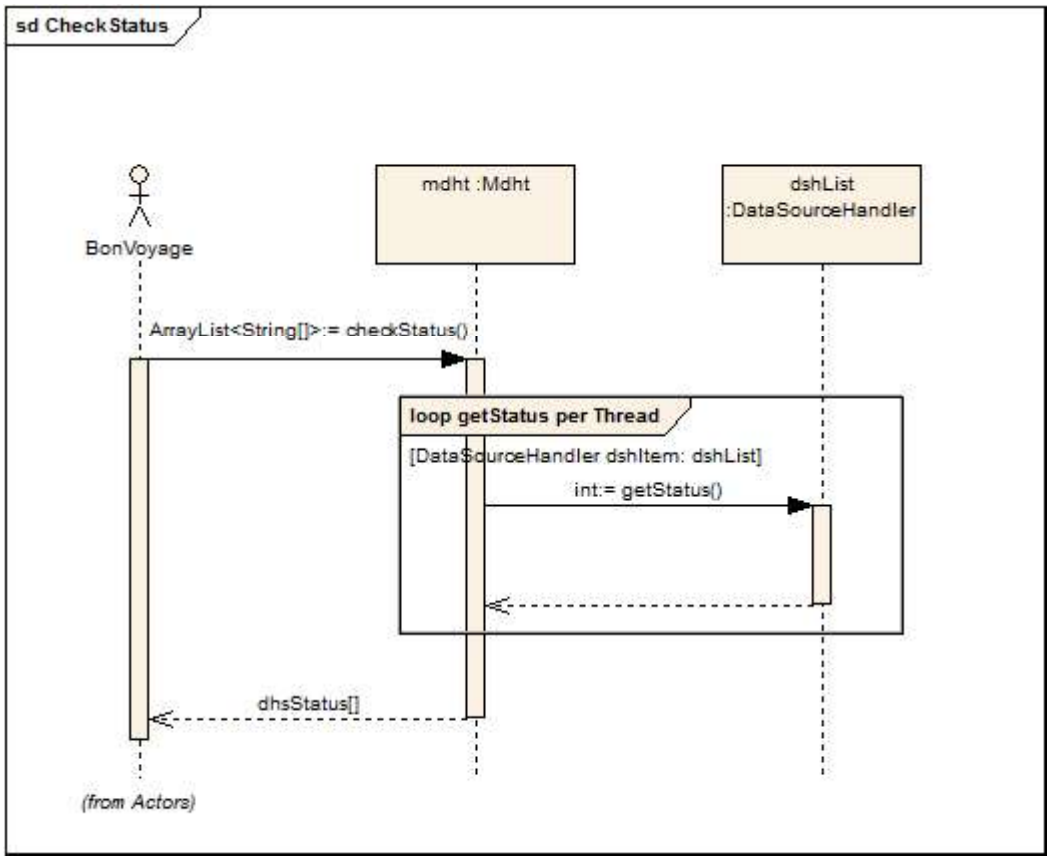

Figure 8: MDHT Check status sequence diagram.

2.2.6.4 Stop global process

The stop process implicates stopping of every Data Source Handler (remind there is one of them for each Data Source). This happens when each of them is in waiting mode.

Figure 9 shows this sequence. The stop order is sent by BONVOYAGE system (or external user) and is received by the main class. Main class, in turn, sends the order to each of the Data Source Handler, and when these are in waiting mode, they stop the process and finish their work, returning control to main class.

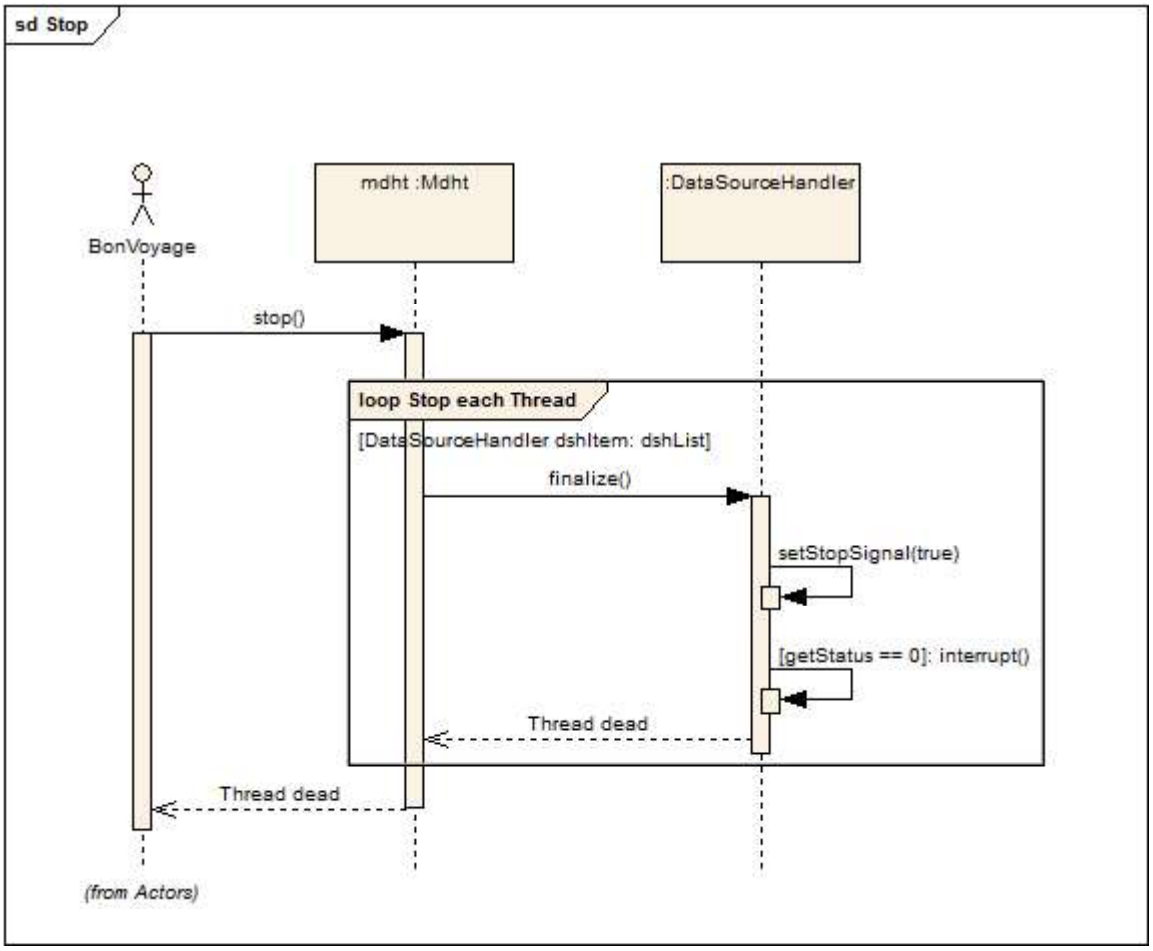

Figure 9: MDHT Stop process sequence diagram.

2.2.7 System design

2.2.7.1 Detailed class diagram

This section of the document explains how the application is developed. Figure 10 shows the class diagram now fully enriched with attributes and methods.

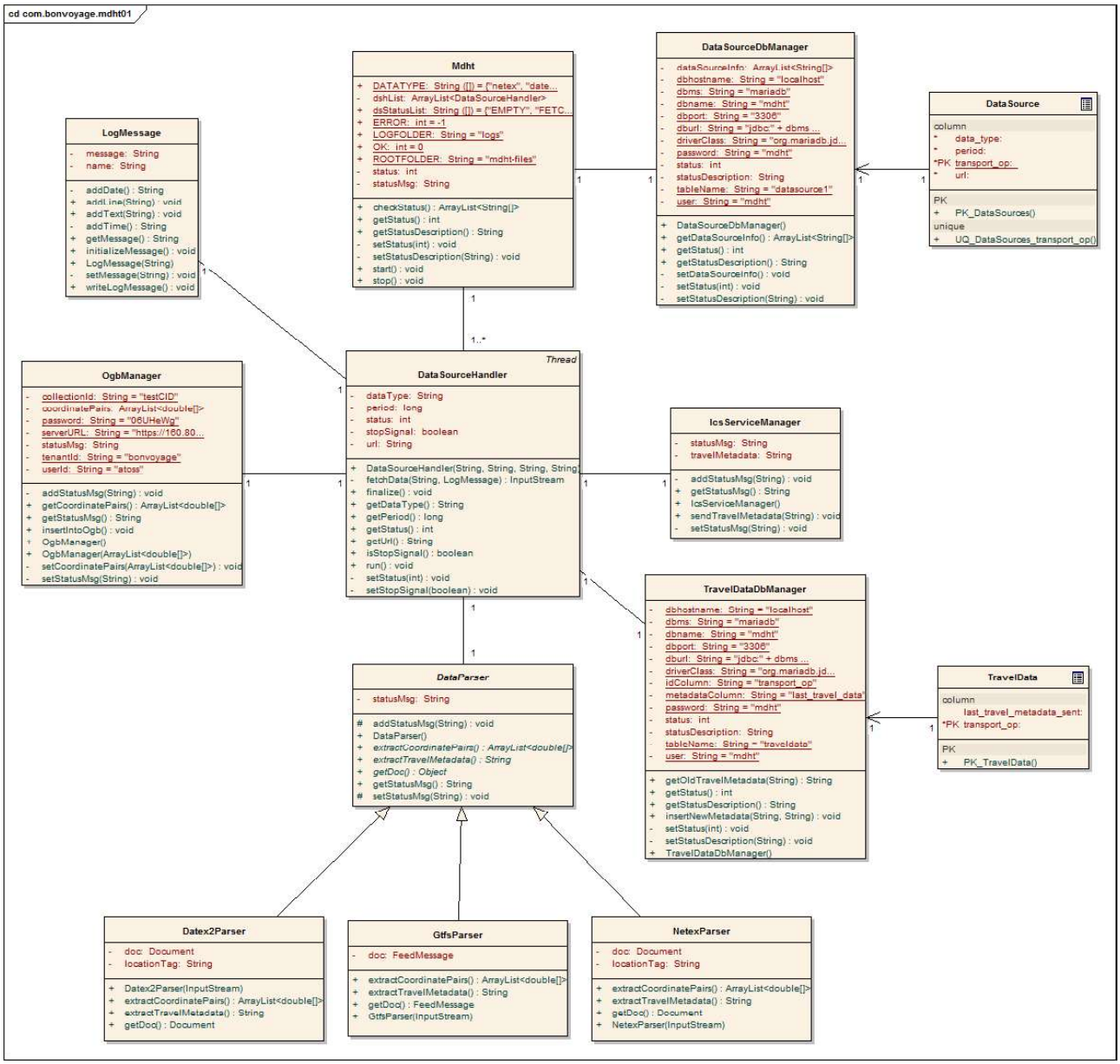

Figure 10: MDHT Class diagram with attributes and methods.

In the diagram, the following classes are implemented:

- **Mdht**. This is the main class of the MDHT application, responsible of starting and stopping global processes, and to provide an interface to external users. These are BONVOYAGE system or external user.
- **DataSourceDbManager**. This class provides a method to get data from the DataSource data table.
- **DataSource** (*database table*). This is a representation of a MySQL database table, which stores configured information about each Transport Operator Data Source like name of the transport operator, URL where the data is available, data type it uses, and the period between data updates.

- **DataSourceHandler**. This class has the main responsibility, because it provides all methods to fetch, process and store the data, for each Transport Operator Data Source. This means that each data source will have a thread assigned to manage it.
- **OgbManager**. This class provides methods to connect and insert GeoJSON for ITS metadata and coordinate pairs into OpenGeoBase and, obviously, to establish connection with it, using its own API.
- **IcsServiceManager**. This class provides methods to send real-time travel data to ICS Publish/Subscribe Service, using its API.
- **TravelDataDbManager**. This class provides methods to get and set data in the TravelData data table.
- **TravelData** (*database table*). This is a representation of a MySQL database table, which stores information about Transport Operator Real-time travel data, and it is used to temporary store the real-time travel data sent to ICS Service, to ease the comparison between them and newer metadata.
- **DataParser** (*Abstract*) This class is a superclass, and it is used to provide an independent way from the data type of Data Sources, to implement operations to analyse data, extract and compare data.
- **NetexParser** (*Extends DataParser*). This class implements some of its father's attributes and methods, in order to specialize in NeTEx format.
- **Datex2Parser** (*Extends DataParser*). This class implements some of its father's attributes and methods, in order to specialize in DATEX II format.
- **GtfsParser** (*Extends DataParser*). This class implements some of its father's attributes and methods, in order to specialize in GTFS format.
- **LogMessage**. This class is the responsible to manage the log writing for each Data Source Handler.

2.2.8 GeoJSON for ITS

GeoJSON for ITS represent the exchange format between MDHT and the OGB instances that are used for discovery purposes.

In deliverable D5.1 we have designed some BONVOYAGE-specific properties of GeoJSON objects, which we use to efficiently represent intelligent transport systems, data sources and services in order to make them discoverable and to easily allow contacting them.

We summarize and extend here this concept of using GeoJSON to represent ITS entities that have a geo-referencing and need to be spatially indexed and searched through a set of concise metadata.

GeoJSON's flexibility comes in the form of a "properties" block that can incorporate custom information about the GeoJSON object.

The following properties block is the one we use in BONVOYAGE to represent ITS-related resources (both services, static data and dynamic data) inside the OGB database, making them searchable and allowing the corresponding resource to be reached/contacted.

The type property

It indicates the nature of the resource being indexed. As of now valid values as FILE, CHANNEL, and SERVICE.

The FILE and CHANNEL types capture the dynamicity of the underlying information. A FILE data source can be expected to change rarely. Its data is thus published as a downloadable compressed feed, bundled as a single file, which is retrieved by a one-shot GET-like operation from the network. It is duty of the user to periodically re-scan the network URL associated with the data source to detect changes. Typically, a FILE data source covers a large geographic area and represents a massive amount of information. The FILE capability is available through a classical HTTP GET scheme of operation.

Conversely, a CHANNEL data source continuously changes in time. It publishes its data as constant feeds under a specific "channel name", to which interested subscribers can attach and receive updates, that is they receive the latest fresh news that refer to that channel as soon as the publisher pushes them and as incremental changes with respect to what was previously received. The publish/subscribe primitives of the BONVOYAGE Internames Communication System are needed to exploit the CHANNEL capability.

The SERVICE type indicates a resource that is dynamic but is going to be queried on demand by interested parties, instead of publishing updates whenever they are available. Thus, a SERVICE is typically contacted through a TCP/IP socket or a RESTful endpoint, for instance.

The sub-type property

It further qualifies the type of the resource.

The sub-type is dependent on the type, and is built around a limited set of strings that are sufficient for the scope of BONVOYAGE. The rationale behind the sub-type is that one should strive to find a sub-type which best matches the nature of a resource, among the already existing sub-types, before attempting to create a new, custom sub-type for the intended resource.

As of now, existing subtypes are:

- The data format, for FILE and CHANNEL types. Example: GTFS, NETEX, DATEXII.
- The nature of the service, for SERVICE type. As of now: ORCHESTRATOR, SOLOIST, CARSHARING.

The url property

It indicates the network endpoint that must be used to fetch data or to contact the service.

The URL representing the network endpoint can be of a different nature according to the type of the source itself: for instance, a name representing a static HTTP URL in the case of a FILE type, or a name representing a dynamic Internames resource in the case of a CHANNEL type, or a TCP/IP endpoint to establish a communication channel with a given soloist, in the case of a SERVICE type.

The name property

It indicates the name of the service operator or the name of the owner of the resource.

For instance, in the FILE type case, for GTFS files, the name field contains the full name of the transit agency, which is the same information stored in the agency-name field of the agency.txt file.

For a SERVICE type, it indicates the institution that operates the Orchestrator or the Soloist, or the name of the car-sharing operator, for instance.

Examples

A static GTFS file of bus schedules can be discovered by representing it with the following GeoJSON for ITS:

Geometry: a MultiPoint where each point is one bus stop

Properties:

type : FILE

sub-type : GTFS

url : http://transitfeeds.com/bus_of_Rome.zip

name : “ATAC bus operator”

A dynamic DATEXII data source publishing Situations about road traffic can be discovered by the following (see D5.1):

Geometry: a Polygon representing a square tile of the territory, pushing all Situations that occur in the area

Properties:

type : CHANNEL

sub-type : DATEXII

url : n2n://bv/8/61/45/36/GPS-ID/datexii/npra/getsituation

name : “NPRA server”

A travel planning service can be discovered by the following:

Geometry: a Polygon representing the area that the planner is able to cover

Properties:

type : SERVICE

sub-type : SOLOIST

url : http://travelplanning.crat.it/rome

name : “CRAT consortium”

A carsharing operator offering vehicles in a certain area can be discovered by the following (see paragraph 2.2.9):

Geometry: a Polygon representing the area the provider operates in

Properties:

type : SERVICE

sub-type : CARSHARING

url : ogb://carsharing/CARSHARING

name : “Car2Go”

2.2.9 Discovery and adaptation of car-sharing services

The inter-modal Soloists that we have developed are able to compute travel solutions that may include car-sharing options as part of the trip. Usually, several different car-sharing operators operate in the same geographic area, and the Soloists would incur a severe performance penalty if they have to inquiry all the different operators in turn, moreover dealing with the different data formats.

The MDHT is able to scan the information provided by the car-sharing operators and:

1) tell the Soloists what operators cover the area of interest, and how to contact them (Discovery);
2) abstract the data coming from different operators in different formats, generating an aggregated view of the several car-sharing offerings which cover the same area, also increasing the Soloist’s performance because one end-point only is going to be called (Adaptation).

Operation 1) refers to the insertion of the coverage area into OGB, so as to make it discoverable by any software wanting to compute travel solutions that include shared vehicles. Once the different operators are discovered and can be contacted, operation 2) refers to fetching, duplicating and normalizing their real-time information into a single and homogeneous table of spatially-indexed vehicles for car-sharing.

Car-sharing operators under scrutiny for test purposes are: SHARE’Ngo; Car2Go; Enjoy. Target area is the city of Rome.

Although all of them publish data about positions of cars and their availability, Car2Go is the only one offering a publicly documented API for querying of data, albeit with some limitations on the query frequency. For the other two, contact end-points were inferred from packet inspection of the communication flows between their Apps and their servers, and from previous studies available at open source repositories (for instance https://github.com/jarek/electric2go).

The following paragraphs 2.2.9.1 and 2.2.9.2 explain Operations 1) and 2), respectively, in more detail.

2.2.9.1 Discovery of car-sharing providers

As stated above, not all car-sharing operators offer an API to retrieve the boundaries of their coverage area, i.e. the area where it is legal to pick-up and leave their vehicles.

Car2Go offers one such API. Its endpoints are:

Locations: provides a list of all locations car2go is operating for

Access: public

URL: http://www.car2go.com/api/v2.1/locations

Method: GET

Operationareas: provides a list of all zones forming the car2go operating area for a given location

Access: public

URL: http://www.car2go.com/api/v2.1/operationareas

Method: GET

Typical response from this last API call is a LineString object representing the perimeter of the operation area, for instance (in JSON format) follows an example “a zone” perimeter where the Car2Go service is excluded:


```
{"placemarks":[{"coordinates":[10.02739,48.458916,0,10.02739,48.468918,0],"name":"a zone","zoneType":"excluded"}]}
```


Unfortunately, Enjoy and SHARE’Ngo do not offer a similar API, thus reverse engineering is needed by means of observing and mapping the coordinates of the points of the perimeter of their coverage area.

In the following Figure 12 a representation of the coverage areas for Rome for Enjoy and SHARE’Ngo is shown.

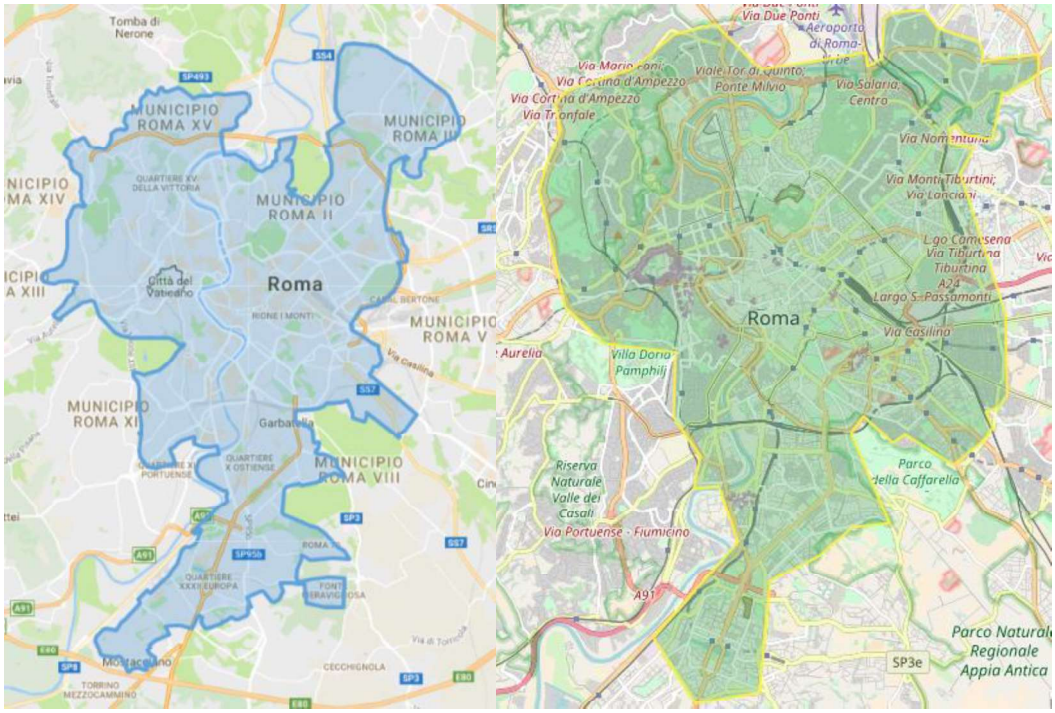

Figure 11: Enjoy (left) and SHARE'Ngo (right) coverage areas for Rome.

The reverse-mapping process of the perimeters of those two coverage zones is ongoing and will be completed during the integration activities of WP7.

2.2.9.2 Efficient access to real-time car-sharing information

As pointed out above, different car-sharing operators store their (publicly available) data in different formats. Hence, we have designed a common subset of crucial information that is needed for planning purposes and available from all providers, which the MDHT is able to extract and store in a cache, in order to simplify the design of any travel planner wishing to include car-sharing offers to the menu of available inter-modal travel solutions.

The cache is:

1) kept in sync with information made available by the operator
2) implemented in a high-performance spatial database
3) designed around a cross-operator schema able to capture the crucial subset of information needed to plan the travels

Regarding point 1) above, we have developed a car-sharing-specific adaptor for the MDHT. It is in charge of periodically sampling data from the various providers, basically "scraping" as much information as possible from the publicly known web services. The sampling frequency is set to be 2/min, under the assumption that the shortest rent is going to be at least a few minutes, thus effectively oversampling the data.

Regarding point 2) above, we decided to exploit our OGB spatial database. Thus, instead of installing another NoSQL server dedicated to host the car-sharing cache, we simply created a new tenant in OGB, in order to effectively separate the "discovery" function of OGB from the "caching" function. The new tenant is a car-sharing dedicated tenant called "carsharing", which contains a single Collection named "CARSHARING".

OGB offers the additional advantage (w.r.t. currently available spatial databases) that, thanks to its native "geographic sharding" design, the cached information about available car-sharing vehicles can be kept within servers that are physically located at network locations "close" to the area the data itself refers to. For instance, if the spatial data recorded in an OGB Collection refers to objects (e.g. vehicles) situated in Rome, it is possible to attach and federate a cluster of servers to OGB that both host that data (together with other data insisting on the Rome area) and are located at network nodes having high-speed connectivity with other Internet services covering the same area or needing to access that data.

Regarding point 3) above, we designed the CARSHARING Collection around a schema that holds the following information in GeoJSON format:

Geometry: a Point representing the current location of a free vehicle;

Properties:

Operator: a string that identifies the vendor/operator of the car-sharing service

OperatorId: a string uniquely identifying the Operator

AvailableFuel: percentage of fuel in the tank or charge of the battery

processedTime: timestamp when the record was sampled.

A single record represents the status of a vehicle that is available for renting.

This kind of OGB-based aggregated cache of information coming from various car-sharing providers guarantees a performance improvement to the Soloists developed by CRAT, which are able to compute inter-modal solutions that include rented vehicles but suffered from a severe performance hit (due to the latency from third-party services) because of the need to contact several different end-points, for each user's request, in order to gather all real-time information about available vehicles and positions.

A noticeable performance increase was achieved by developing this car-sharing MDHT adaptor. Further details will be given in upcoming D7.x deliverables where integration is described in details.

2.3 Implementation of the Trip Planning Services

Design aspects of the architecture and algorithms for the trip planning services were reported in deliverable D4.1. The algorithms were implemented into trip planning services as reported in deliverable D5.1 and algorithms were tested for applicability and optimality as reported in deliverable D4.2. We will now provide further insight into the current implementation.

2.3.1 Implementation of the Orchestrator Service

The current orchestrator is implemented in the C# object-oriented programming language using the .net framework 4.5 and the Windows Communication Foundation (WCF) APIs to fit into the BONVOYAGE service-oriented architecture. Table 15 lists the classes that together make up the implementation of the core orchestrator. In addition, to realize the orchestrator as a service, we have implemented the orchestrator in a web service hosting application which also communicates with the discovery service. To allow for vertical scalability of the orchestrator, the hosting application utilizes *instance pooling* for the orchestrator.

Class	**Responsibility**
CostDimension	Enumerates the different dimensions in which the cost of a travel can be measured. Each dimension has a predefined unit.
DestinationNode	Represents a destination of a trip request and is inserted into the orchestrator graph before the orchestrator processes the request.
IOrchestratorGraphItem	Shared interface for both *IOrchestratorArc* and *IOrchestratorNode*.
IOrchestratorArc	Interface for an arc in the Orchestrator graph.
IOrchestratorNode	Interface for a node in the Orchestrator graph.
IRouteSolver	A route solver for any BONVOYAGE routing service. In the case of an orchestrator it is implemented by *OrchestratorBase*. In the various cases of soloist instances, it wraps the specific soloist's algorithm or API.
Modality	Enumerates the possible modes of transport used for travelling.
ObjectivePoint	Stores a single observation of the objective for a specific set of weights for a list of *CostDimensions*.
ObjectiveStatus	Enumerates the possible status values for an *OrchestratorGraphObjective*.
Orchestrator	The current implementation of a full orchestrator. It contains an *OrchestratorGraph* to represent the relationship between soloists.
OrchestratorBase	Abstract base class for Orchestrators and includes properties and methods necessary for all implementations of orchestrators. It implements the *IRouteSolver* interface.
OrchestratorArcBase	Base class for the different instances of orchestrator graph arcs.
OrchestratorGraph	The graph used by the Orchestrator to generate requests to the soloists. The graph is designed to be solved using a label setting algorithm to find a shortest path through the network. The algorithm takes an objective and can have a set of resources or constraints. The routing algorithm requires that all arcs hold a non-negative contribution to the objective.

OrchestratorCachedGraph	Responsible for caching of previously computed *SoloistArc* objectives to be read and reused between sessions.
OrchestratorGraphLabel	Labels representing intermediate costs and route path information used to solve a routing request efficiently during a search in the orchestrator graph.
OrchestratorGraphObjective	Each *IOrchestratorGraphItem* has an *OrchestratorGraphObjective* that gives the lower bounds. It also maintains a list of *ObjectivePoints* that are used to calculate the lower bound.
IOrchestratorTransition	Interface for a transition point, either *OrchestratorTransitionPoint* or *OrchestratorTransitionArea*. Two transitions having different soloists and that overlaps geographically and in modalities can be connected through a *TransitionArc*.
OrchestratorTransitionPoint	Represents a potential transition between soloists, has a specific geographical coordinate.
OrchestratorTransitionArea	Represents a potential transition between soloists and covers an area.
OrchestratorNodeBase	Abstract base class for the different instances of orchestrator graph nodes.
OrchestratorPath	An orchestrator path manages the response from the orchestrator graph. It contains a list of orchestrator path items, that holds the requests for the soloist services and their responses.
OrchestratorPathItem	Contains the request to the soloist as *OrchestratorSoloistRequest* and the corresponding response as SPROUTE RouteFormatRoot.
OriginNode	Represents a start location of a trip request and is inserted into the orchestrator graph before the orchestrator processes the request.
Soloist	A soloist within the concept of an orchestrator. It knows about the service signature (URL etc.) provided by the discovery service and the characteristics provided by a previous request to the soloist itself.
SoloistArc	An arc internally within a soloist in the orchestrator graph which connects two of the soloist's transitions. Traversing the arc contributes with an additional cost in the overall objective value of a route.
SoloistCharacteristics	Stores the characteristics for a soloist. It contains a Boundary for the soloist, a list of Modalities that can be supported, a list of transitions at which the soloist can be connected to other soloists, and a list of modalities are associated with these points.
SoloistResponse	Contains the resulting routes of a route request returned by a soloist routing service.
SoloistRoutingService	The orchestrator's client side for a soloist, taking care of all connection and communication issues

TransitionArc	An arc between transition nodes in two different soloists in the orchestrator graph. This arc is normally not generating any cost or time delay.
TransitionNode	A node in the orchestrator graph. A *TransistionNode* is connected and owned by a soloist. Two *TransitionNodes* having different soloists can be connected when they overlap geographically and in modality.

Table 15: Classes in the implementation of the orchestrator.

The algorithm of the orchestrator for processing trip requests is as follows:

1. The service hosting application receives a request in SPROUTE serialized format and de-serializes it.
2. It then calls IRouteSolver.CalculateRoutes, e.g., implemented in class OrchestratorBase which again calls the abstract method OrchestrateCalculationOfRoutes, e.g., implemented in class Orchestrator.
3. The method OrchestrateCalculationOfRoutes does the following
    a. Modifies the graph to accommodate the from-location of the request as a source and the to-location as a sink.
    b. Calls the multi-objective shortest part algorithm to find the shortest path from source and sink.
    c. For each OrchestratorPathItem in the shortest path it calls the corresponding soloist with a sub-request to get the sub-route across the soloist. Time-independent soloists are called in parallel while time-dependent soloists are called sequentially.
    d. Compares the sub-responses with the lower bounds it has stored in the graph and, if there is no gap, terminates. Otherwise the orchestrator updates the lower bounds and repeats steps b to d. Through this repetition, it learns more and more about the behaviour of the soloist. Hence, the orchestrator becomes better and better in knowing which soloists to include the more requests it receives.
4. Finally, the orchestrator assembles the sub-responses into a complete response and returns it to the hosting application which serializes it and sends it to the client from which the request originated.

2.3.2 Implementation of the Soloist Services

We have created two instances of soloist services, which are implementing the IRouteSolver interface described in section 2.3.1. On the other side, the Soloist Services designed and developed by CRAT in Task 4.2 concern urban and peri-urban planning systems. These services have been implemented as PHP web services exposing RESTful APIs in order to be easily integrated in the overall BONVOYAGE system as detailed in D7.1. The implementation has already been completed and reported in WP4 (see D4.2, section 5.2 “Urban soloist”), where three distinct applications have been described and detailed:

- Bilbao Soloist: The Soloist Services cover an area of about 15Km x 10Km around Bilbao city centre;
- Rome City Centre Soloist: The Soloist Services consider exclusively the Rome city centre for an area of about 12Km x 7Km;
- Rome Soloist: The Soloist Services cover all the urban area of the city for an extension of about 25Km x 25Km.

All details regarding maps and data structures, as well as implementation details, have been reported in D4.2 and are not duplicated here for sake of brevity. Please refer to deliverable D4.2 for a complete description of implementation of the Soloist Services.

The instance *RouteSolverDynamo* utilizes *DYNAMO* which is SINTEF's multi-objective and multi-modal trip planning optimization library[1]. The adaptations necessary for being used in the BONVOYAGE project were merely to align the cost dimensions of the DYNAMO objective to the cost dimensions used in BONVOYAGE. Dynamo provides optimal intermodal trips that can include any combination of the following modalities:

- Pedestrian
- Bicycle
- Car
- Wheelchair
- Motorcycle
- Public transportation, i.e., all a-priori scheduled public transportation

The relevant properties of a traveller are stored in a traveller object, i.e., the individual travel preferences and CO_2, emission, travel fare, and energy consumption parameters when using a given modality. The energy consumption parameters are for instance relevant to obtain the correct travel speeds then travelling with a bicycle, taking into consideration the energy input, friction, drag and road incline. The travel fare parameters make it possible to better estimate the ticket and toll-road prices applicable for the traveller in question. A single trip request to DYNAMO may result in multiple route responses, each representing unique combinations of the modalities, that are allowed according to the request. The test results reported in section 3.2.3 are based on soloists using *RouteSolverDynamo*.

The instance *RouteSolverGoogle* is implemented to illustrate how existing web services can be incorporated into the BONVOYAGE framework. In this case, we are using Google's Directions API[2] "Modes of Transport" in the following way: When sub-requests in SPROUTE format are received from the orchestrator, they are converted into REST service URL strings. We pass these strings to a WEB-request GET method. The resulting web stream is read as an XML document, which is parsed to build a SPROUTE response that is returned to the orchestrator.

2.4 Adaptations of the trip planning services for scalability

The orchestrator is a key component for allowing for horizontal scalability (by adding more soloists) and vertical scalability (by being able to fully exploit the available computational resources). Therefore, in addition to complete the implementation reported in section 2.3, we have made several adaptations to ensure a high utilization of the computational resources (CPUs) available. The capacity of the computational resources can be better utilized by means of:

- Introducing redundant trip planning services.
- Computers/cloud servers with multiple CPUs, each potentially having multiple cores

[1] The development of DYNAMO preceded the BONVOYAGE project, and, as stated in the BONVOYAGE Consortium Agreement, SINTEF will keep the ownership and the copyright of DYNAMO including any updates made during the project.
[2] https://developers.google.com/maps/documentation/directions/

The first case may happen when new trip planning services (soloists) become available which have similar characteristics and covers the same geographical area as existing soloists. In this case, we want the orchestrator to distribute sub-requests between the different redundant soloists to maximize the utilization of each services and hence to minimize the overall response time. The adaptation to the orchestrator is then to allow a SoloistArc to address more than one soloist, and forward the sub-requests to the soloist that at the moment is expected to have the fastest response time. A nice feature of adding the possibility of redundant trip planning service is that soloist services can be added and removed without restarting the orchestrator service, e.g., when a server hosting a soloist needs to be restarted for service maintenance, a soloist service can be started on a different server before the server is restarted. Hence, there is no time window for which the service is unavailable from the perspective of the BONVOYAGE users.

The second case is applicable to both the orchestrator services and the soloist services. These services can be implemented in a multithreaded fashion, allowing multiple tasks to utilize the service data and service object simultaneously and in this way process multiple requests simultaneously. An alternative way is to use instance pooling. This is an easy technique that also enables non-multithreaded services to process multiple requests at the same time, but at the expense of increased memory usage. To understand instance pooling, it is first important to note how, at least in WCF, a web service request is processed. The web service receives a request and needs somehow to get an instance of the object that can execute the request, in our setting the trip request. The object can e.g. be generated on the fly, or it can be taken from an instance pool. The latter is preferable if the generation of such an object takes a long time. This is the case for both orchestrator and the soloists used by SINTEF based on Dynamo as described earlier. In both cases an object creation involves reading data and generating a graph, which takes a considerable amount of time. With instance pooling, this time is moved to the start-up of the service. Once the service is up and running, it will have an instance pool of objects ready to be used for serving incoming requests. The web service can be configured such that each object is only used by one thread at a time to exclusively handle one request at a time.

In the experiments performed by SINTEF in section 3.2 we utilized the instance pooling approach for both the orchestrator and the soloists.

3 Tests of components

In addition to the test results reported in previous deliverables, we have tested the BONVOYAGE sub-components in various ways to ensure sufficient stability and performance of each component before we test the full integration of the complete system. Those tests are reported in this section.

3.1 Tests of the Metadata Handling Tool

3.1.1 Unit tests

It is known that *unit testing* is a software testing method by which individual units of source code, sets of one or more computer program modules together with associated control data, usage procedures, and operating procedures, are tested to determine whether they are fit for use.

In the case of the Metadata Handling Tool, these tests have been done throughout their development process, testing each method and each class. Although ideally, each test case is independent from the others, some classes may have references to other classes or databases, so the best way to prove that the application meets the functional requirements defined previously, is testing several classes, and finally, the whole application.

The tool used in MDHT to test the code and complete functionality has been Junit4, since the application has been developed using Java, with Eclipse. Throughout the process, the classes and methods that were being implemented have been tested, providing inputs and checking outputs. To verify the functionalities, the complete application was tested, checking that possible errors in the data sources were controlled, and that the outputs were the expected ones at each step of the process.

Unit tests for small parts of code have affected the following classes and methods:

- *DataSourceDbManager* and *TravelDataDbManager*, as database interfaces, tests that have been performed were connection, read and write data, and close, controlling the cases of impossibility of doing any of them.
- *OgbManager* and *IcsServiceManager*. As they are implemented as processes responsible of writing or sending information exclusively, the tests have included the processes of connecting and sending information, waiting for a response by the destination, as well as causing communication errors to verify that they are checked and managed.
- *LogManager*. This class is directly related to the main ones. However, as it performs writing on files of the server, collecting the information of the processes, tests have been made considering the different situations that can arise accessing the local filesystem.

The approach we have followed in order to test the complete behaviour has been that of driving the testbed by means of an ad hoc application that triggered the complete set of functionalities. This test particularly affected the most relevant classes of the MDHT: *Mdht* and *DataSourceHandler*. They have been tested with different types of data sources and different situations, and by qualitatively checking that the message flow and data output behaviour of the application was as expected.

3.1.2 Stress test of single components

We performed stress tests mainly by running the process against a high number of sources of data from which to gather information.

As the application is designed to divide the work into as many threads as there are data sources, the performance of the application depends directly on the power of the machine on which it runs and, above all, the connections to the data sources.

One of the ways in which the system could be improved would be to divide the work of sending or registering information into OpenGeoBase and ICS Service into two separate threads, although in the end both would have to be finished before the process can sleep for the corresponding period. This would imply that each thread in each data source was divided into two.

3.1.3 Horizontal scalability

Neither large growth of the number of OpenGeoBase objects, nor big volume of data transported by ICS Services do affect the operation of the MDHT as a standalone component, and they are not controlled by it. However, testing of its integration with the rest of BONVOYAGE components interacting with it will be part of activities of D7.x and of the integration plan.

3.2 Tests of the Trip Planning Services

SINTEF did already in deliverable D4.2 report some results from testing of the orchestrator and trip planning services, see sections 5.1.3 and 5.2.2 in D4.2. Those tests verified the optimality of the resulting routes in the case of static travel times on the road, i.e., without considerations with respect to rush-hour traffic or real-time road traffic incidences. We also reported how the response times for individual trip requests scales up very well for long trips. The longest trips tested were about 2700 km on the road, which were planned with response times to the client in about 1 second.

On the other hand, several tests of the Trip Planning Services represented by the Soloist Services developed by CRAT have been performed in Task 4.2 for three applications mentioned in section 2.3.2: Bilbao Soloist, Rome City Centre Soloist and Rome Soloist. All the test instances have been processed by an Apache server hosted in a cloud server (provided by the Cloud Service Provider *1&1*[3]) running Linux Ubuntu 16 with 16 virtual CPU, 8GB RAM and 120GB of SSD. A complete and exhaustive overview of the tests, including description of instances and discussion about results, from both the effectiveness and efficiency point of view, has been reported in D4.2. The considered Key Performance Indicators (KPIs) included: Average number of test problems, Average number of distinct solutions returned by the Soloist Services, Average planning time, Average computation time and Average End-to-End time. Unit tests and setup of the performance tests have been reported in section 5.2.1, while results have been summarized in section 5.2.2 of D4.2. Deliverable D4.2 has been submitted by June 2017 according to project activity schedule.

What SINTEF did not test in D4.2 is how the response times are affected when high number of simultaneous trip requests are sent to the orchestrator service simultaneously, i.e., load stress performance tests of the trip planning

[3] https://www.1and1.com

service components of the overall BONVOYAGE architecture. We have therefore created an environment for performing such tests, and the results are reported later in this section.

3.2.1 Unit tests

A unit test is a test of some unit, such as a function or a class. We have applied unit tests for each unit of components at the level of individual functions or classes. Ideally a unit tests does not rely on code in the tested project outside the tested unit. The software for trip planning has been extensively covered by unit tests to ensure stable results as the code has been subject to changes. As can be seen in Figure 12, 72,4% of the source code for the orchestrator algorithm is currently covered by miscellaneous unit tests. While unit tests do not test the execution performance of the code, it has been found extremely useful to shorten the development cycle and maintain a coherent code.

	Code Coverage %	Compiled Lines	Covered Lines	Uncovered Lines	Code Lines	Excluded Lines
Orchestrator	72,40 %	2 203	1 595	608	5 131	0
CostDimension.cs	100,00 %	18	18	0	93	0
DestinationNode.cs	100,00 %	3	3	0	22	0
IOrchestrator.cs		0	0	0	101	0
Modality.cs	100,00 %	18	18	0	87	0
Orchestrator.cs	61,63 %	516	318	198	884	0
OrchestratorArcBase.cs	95,45 %	22	21	1	102	0
OrchestratorBase.cs	63,34 %	371	235	136	802	0
OrchestratorCachedGraph.cs	53,49 %	43	23	20	126	0
OrchestratorGraph.cs	95,29 %	361	344	17	744	0
OrchestratorGraphLabel.cs	62,79 %	86	54	32	202	0
OrchestratorGraphObjective.cs	84,08 %	201	169	32	456	0
OrchestratorGraphTransitio ...	47,37 %	57	27	30	252	0
OrchestratorNodeBase.cs	76,67 %	30	23	7	113	0
OrchestratorPath.cs	60,50 %	119	72	47	272	0
OriginNode.cs	100,00 %	3	3	0	22	0
Soloist.cs	87,20 %	164	143	21	321	0
SoloistArc.cs	100,00 %	9	9	0	54	0
SoloistCharacteristics.cs	65,09 %	106	69	37	247	0
SoloistRoutingService.cs	53,45 %	58	31	27	147	0
TransitionArc.cs	100,00 %	4	4	0	29	0
TransitionNode.cs	78,57 %	14	11	3	55	0

Figure 12: Unit test coverage for the orchestrator source code.

3.2.2 Setup of the load stress performance tests

A separate client has been developed for executing load stress tests of the trip planning services. It works in the following way:

- It loads a set of coordinates from file. In Figure 13 the six coordinates used in the experiment are marked as small red circles. The average road distance between these locations is 170 km.
- We can specify the total number of trip requests to be sent. The start and end coordinate for the trip requests are selected by looping through the coordinates loaded. Hence, if there are 6 coordinates and the number of trip requests is 30 or more, all combinations of trip requests are made. In our experiments, we used from 30 to 1000 trip requests.
- We can specify if trip requests are to be sent sequentially or asynchronously (i.e., possibly in parallel).

- For the asynchronous experiments, we specify the duration of the time interval within which trip request are to be sent. Each trip request is sent at a random time (uniformly distributed) within this time interval. In our experiments, we set the interval to be 60 seconds.
- During the execution of the test, the client (and the services as well) logs data necessary for later analysis.

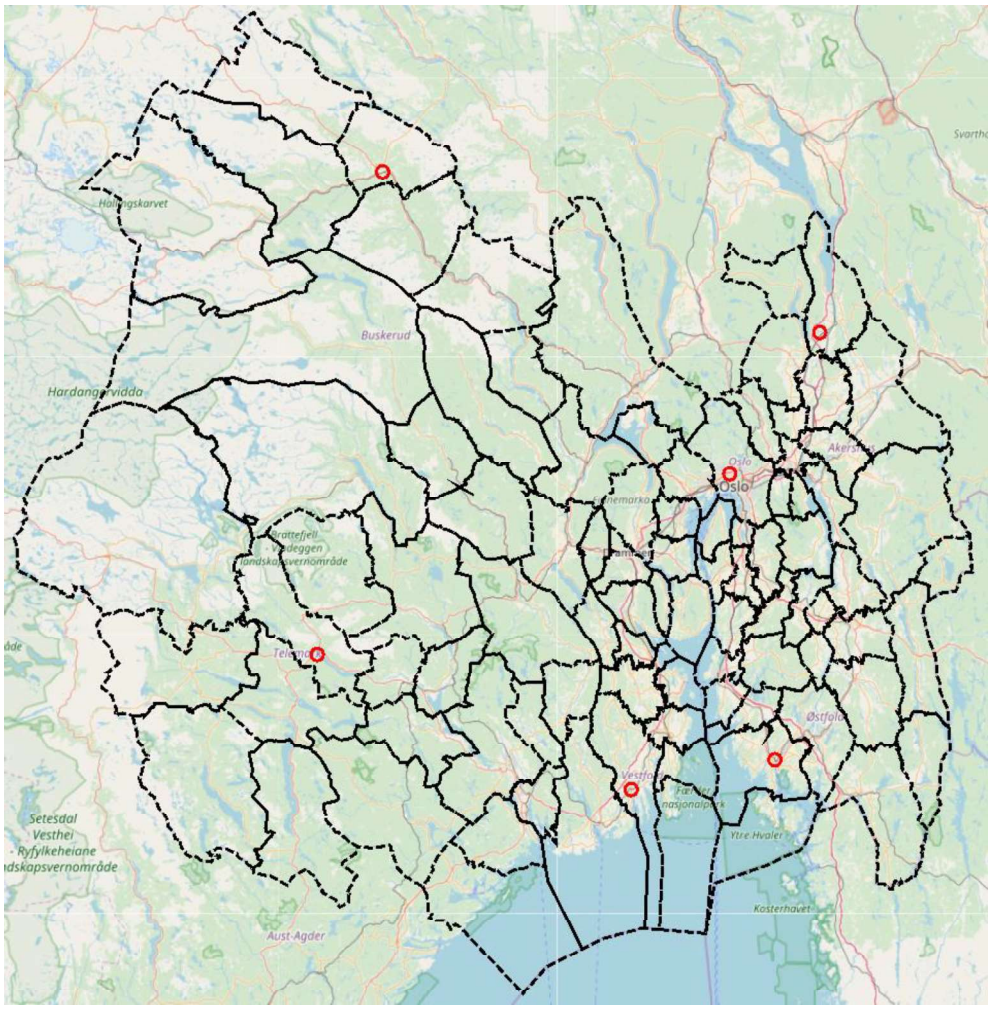

Figure 13: Geographical area covered by the 92 soloists used during load stress tests.

We used the server at http://bonvoyage.sintef.no/ to host both the orchestrator and soloist the trip planning services. Figure 13 shows the 92 soloists used in our load stress performance test. Together they cover six counties in the south-eastern part of Norway. The server is configured with 64 GB physical memory, and with 4 processors each having 4 cores. As a reference for later figures, Figure 14 shows the activity of the 16 cores right before we executed our experiments.

Figure 14: Utilization of the 16 CPU cores at near idle state.

3.2.3 Results of the load stress performance tests

3.2.3.1 Reference scenario: Sequential requests

In the reference scenario, we sent 600 trip requests sequentially to the server, i.e., as soon as the client received a trip response, it sent the next trip request. The total duration of the experiment was 6 minutes and 54 seconds, with an average response time of 0,7 seconds. All responses were returned well within 2 seconds except one which was returned after 11 seconds. The likely reason for the deviating result is due to the request causing the need for vertical scaling of a soloist, which it had to wait for. From Figure 15 we can see that the available processing capacity was not well utilized, mostly allocating the tasks to the first core of each CPU (top row in Figure 15).

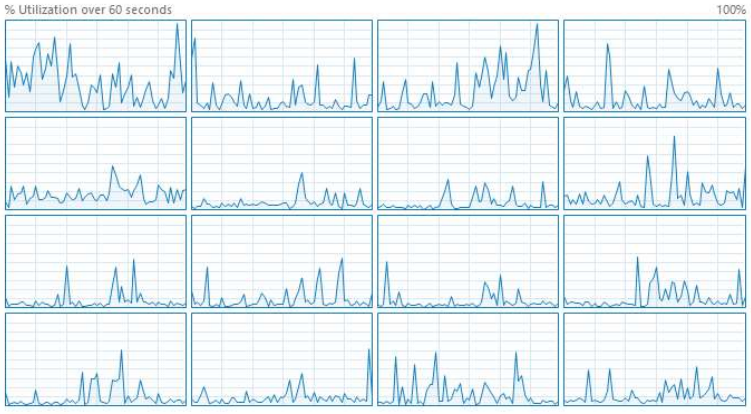

Figure 15: CPU utilization during sequential processing of trip routing requests.

3.2.3.2 Scenario using multiprocessing services

In this scenario, we used the adaptations for vertical scaling explained in section 2.3.1. We sent all 600 trip requests to the server at random times within 60 seconds while monitoring the CPU utilization. As we had hoped, the processing capacity was well used, and the last response was returned 1 minute and 45 seconds after the start of the experiment, i.e., almost 4 times faster than the reference scenario. The requests towards the end of the experiment took up to 48 seconds to process. Hence, sending an average of 10 trip requests per second was above the capacity of the server.

We therefore made a sequence of experiment runs where we sampled the number of trip requests within 60 seconds, in the range between 30 and 1000. The results are summarized in Figure 17, where we see the total, average and maximum processing times for each run. For the runs with up to 400 request per minute, the maximum response time (typically happening for requests towards the end of the experiment) were processed just as fast as other requests. Hence, no backlog queue of requests to be processed was built during this experiment. From this we can conclude that the capacity of this particular setup, before the response time becomes unacceptable, is around 6 requests per second. We can also conclude that vertical scaling works and is easy to implement.

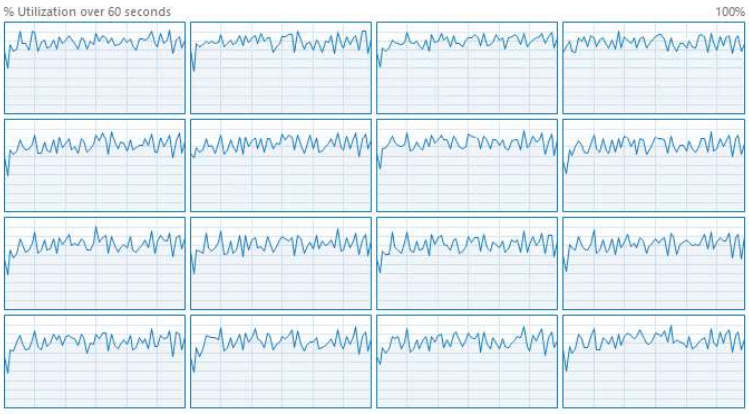

Figure 16: CPU utilization during parallel processing of trip routing requests.

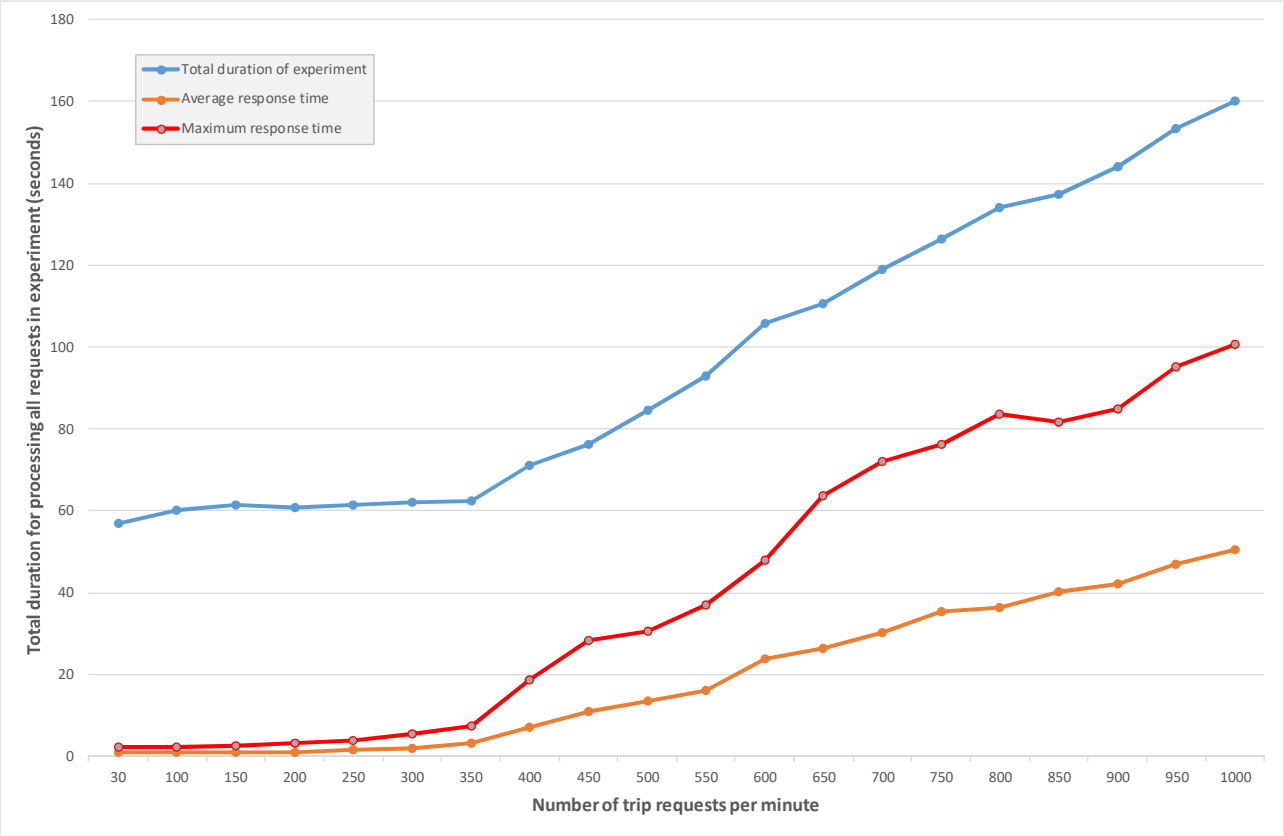

Figure 17: Processing times at different load levels.

Figure 18 shows a heat map coloured according to the peak number of requests processed in parallel by each soloist. The highest number of necessary soloist instances is for the soloist of Oslo Municipality (16), Sandefjord (6), Horten (6), Re (6) and Eidsvoll (5). That the Oslo soloist needed the highest peak capacity for processing requests is expected due to its central location with respect to major roads, and because the processing of each request takes a longer time then for other soloists due to the size of the road network. Of the 92 soloists, there were 47 soloists for which only one instance was sufficient, and 46 of these 47 soloists were never used for the specific combination of trip requests in our experiments. Consequently, only 37 GB of the 64 GB of RAM available on the server was used.

This experiment is a non-regression test that demonstrate the advantage in the BONVOYAGE distributed approach for trip planning in how the limited capacity in processing and memory can be dynamically allocated to "hotspots" where the demand for computation is the highest. If we had used an approach where all capacity had been adjusted for the peak demand of a smaller area, the server would have run out of memory even for this limited part of Norway (only 92 out of Norway's 426 municipalities).

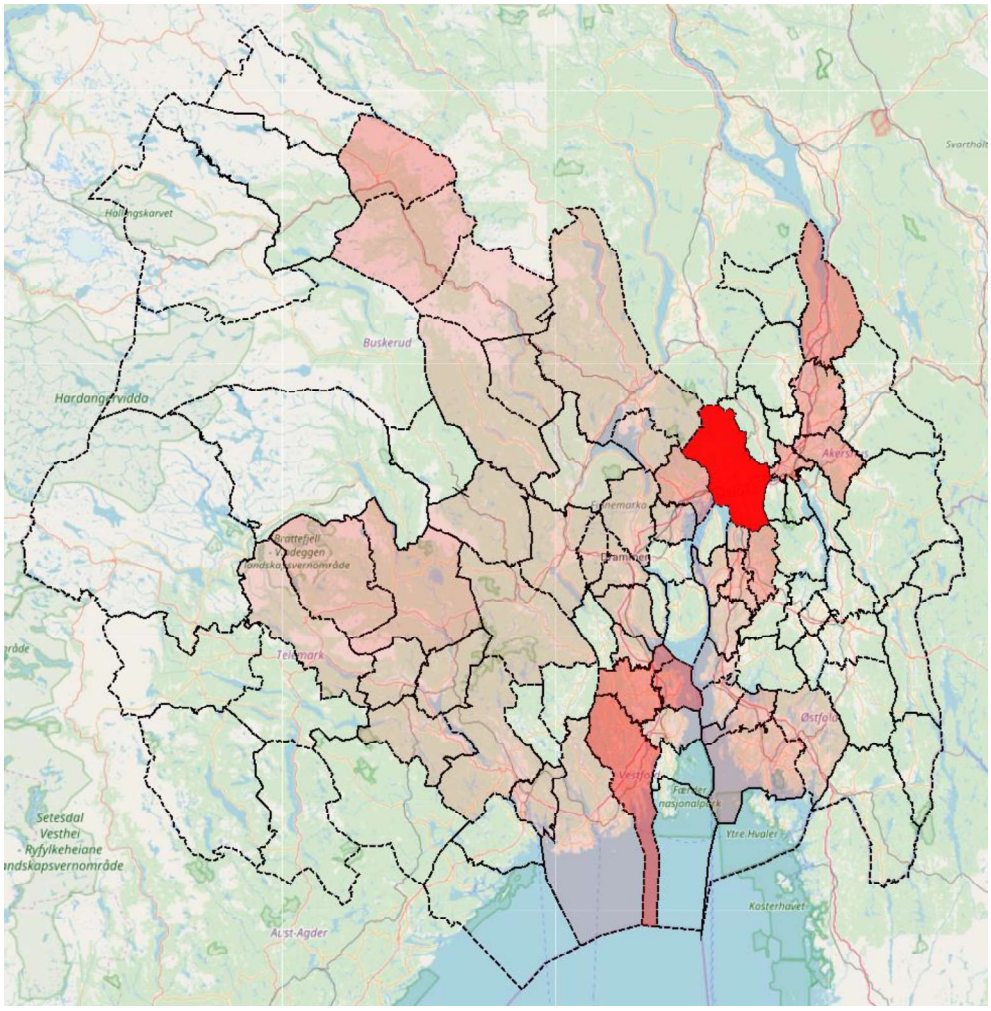

Figure 18: Heat map showing the peak number of requests processed in parallel by each soloist.

4 Software repository

In addition to this document, the following software is made available as integral part of D5.2:

Component	Sub-component	Made available in SVN at[4]	Open source
MMMDB	Dump file of MMMDB	WP5_Adaptation functionality/Software/MMMDB	No
MDHT	Metadata Handling Tool reference implementation	WP5_Adaptation functionality/Software/MDHT	No
	Car sharing and Bilbao CoCities adaptors	WP5_Adaptation functionality/Software/MDHT	No
	DATEII adaptor	WP5_Adaptation functionality/Software/MDHT	No
Orchestrator	Orchestrator algorithm	WP5_Adaptation functionality/Software/Orchestrator	No
	Service interfaces	WP5_Adaptation functionality/Software/Service interfaces	No
Soloist	Service interfaces	WP5_Adaptation functionality/Software/Service interfaces	No
	RouteSolverGoogle	WP5_Adaptation functionality/Software/Soloists	No
	Urban Soloists	WP7_System integration and validation[5]	No

Table 16: Software components

[4] The SVN is found at https://minerva.netgroup.uniroma2.it/svn/bonvoyage.

[5] Since the three implementations of the Urban Soloist designed and developed by CRAT in WP4 have built-in adaptation functionalities, the developed software (as reported in D4.1 and D4.2) and the related interfacing code will be provided in the work package dedicated to system integration, namely WP7.